Improved Shortest Path Algorithms by Dynamic Graph Decomposition

---

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Master of Science
in
Computer Science and Software Engineering
in the
University of Canterbury
by
Lin Tian

Professor Tadao Takaoka	Supervisor
Dr. David J. Pearce (Victoria University of Wellington)	Examiner

University of Canterbury
2006

To my parents and wife.

Abstract

In this thesis, we introduce three new approaches for solving the single source shortest path (SSSP) problem in nearly acyclic directed graphs, and algorithms based on these approaches.

In the first approach, we extend a technique of strongly connected components (sc-components) decomposition by Takaoka [23], and the generalized decomposition approach is called a *higher-order decomposition*. According to Takaoka's definition of acyclicity, the degree of cyclicity of a graph G, $cyc(G)$, is defined by the maximum cardinality of the strongly connected components of G. Based on the *higher-order decomposition*, we give a generalization of Takaoka's definition of acyclicity. That is, the degree of cyclicity $cyc^h(G)$ is the maximum cardinality of the h^{th} *order* strongly connected components of G, where h is the number of times that the graph has been decomposed. Then, the original definition introduced by Takaoka [23] can be presented as: The degree of cyclicity $cyc(G)$ is the maximum cardinality of the 1^{th} *order* strongly connected components of G.

The second approach presents a new method for measuring acyclicity based on modifications to two existing methods. In the new method, we decompose the given graph into a 1-dominator set, which is a set of acyclic sub-graphs, where each sub-graph is dominated by one trigger vertex. Meanwhile we compute sc-components of a degenerated graph derived from triggers. Using this preprocessing, we can efficiently compute the single source shortest paths (SSSPs) for nearly acyclic graphs in $O(m + r\log l)$ time, where r is the size of the 1-dominator set, and l is the size of the largest sc-component.

In the third approach, we modify the concept of a 1-dominator set to that of a 1-2-dominator set, and achieve $O(m + r^2)$ time to compute a 1-2-dominator set in a graph. Each of acyclic sub-graphs obtained by the 1-2-dominator set are dominated by one or two trigger vertices cooperatively. Such sub-graphs are potentially larger than those decomposed by the 1-dominator set. Thus fewer trigger vertices are needed to cover the graph, that is, $r' \leq r$, where r' is the number of triggers in the 1-2-dominator set. When r' is much smaller than r, we can efficiently compute SSSPs in $O(m + r'\log r')$ time.

Table of Contents

Acknowledgments

I would like to thank my thesis supervisors Professor Tadao Takaoka and Dr. Brent Martin for their guidance and assistance with this thesis, and I am especially grateful to Professor Takaoka for providing me with this opportunity. I thank the examiner David J. Pearce from the Victoria University of Wellington for reviewing and marking this thesis.

I would also like to thank Bae Sung for his help. He took the time to answer my questions and guided me through the research.

Both Bashar Mohammad and Ray Tan deserve huge thanks for creative discussions with me which have inspired my research.

Finally, my thesis research was supported by my parents and my older brother. I thank them for their supports and encouragements. Without their financial support, my decision to come to New Zealand would have been much more difficult.

Chapter I

Introduction

Human beings can naturally solve the shortest path problems. When we drive from home to work, we may think of the shortest route from home to office because it can save our time and petrol. Thus, the shortest path problem is referred to as a problem of efficiency. Generally speaking, finding the shortest path is to minimize the cost. If we need to find the most cost-efficient route through a transport system in a city like New York, or a communication network with a large amount of connections like the Internet, then we will need more advanced technology to solve the shortest path problem instead of human intuition [8].

People use graphs to model problems of the real world. A graph is defined by a set of *vertices* and a set of *edges* that connect these vertices (see Figure 1.1). Therefore, to interpret a transport system of a city into a graph, the buildings can be modeled as vertices, and the roads linking them are edges in the corresponding graph. To model a computer network, in the corresponding graph the vertices can represent the computers in the network. The edges can be the communication links between computers [21].

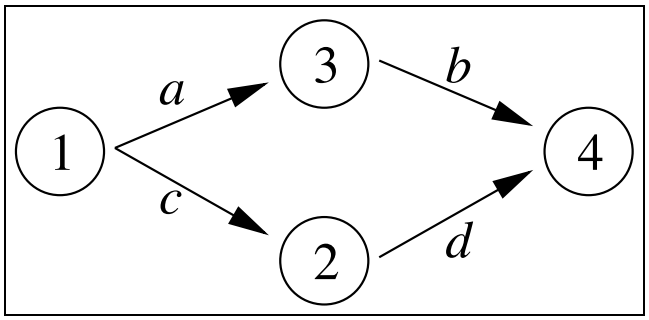

Figure 1.1: An example of directed graph consisting of vertices $1, 2, 3, 4$ and edges a, b, c, d

An edge is specified by two vertices (*end-points*) that it connects. In Figure 1.1, edge $a = (1, 3)$. A path in the graph is a sequence of edges $(v_0, v_1)(v_2, v_3), ..., (v_{k-1}, v_k)$, and we call that a path from v_0 to v_k. If $v_0 = v_k$ and $k > 0$, the path is called a *cycle*. If each edge has a cost, the sum of these costs is the cost of the path. If there is more than one path between two vertices in a graph, then it is possible that one path has a smaller sum of costs than that of another path. This raises the problem of finding the shortest path in a graph [4].

To find the shortest path through a graph, we repeat adding up costs for each path and compare the sum of costs to find the minimum. That is to say, a shortest path problem can be solved by following a repeatable list of steps. Such a list of steps is called an *algorithm* [15]. Thus, a shortest path algorithm is a list of steps that shows how to calculate the shortest paths. There are different algorithms for solving the same problem. One algorithm may be more efficient than another in terms of solving the same problem. An algorithm can be implemented in computer languages like C language. Such a computer implementation is called a *computer program*. A program can process huge graphs that human beings cannot handle by hand. A program based on a more efficient algorithm uses less time to solve the same problem than programs based on less efficient algorithms. Thus, when a program has large input data, an efficient algorithm will have significant influence on the performance of the program in terms of time consumed [21].

When we need to determine the shortest paths quickly or repeatedly for a large database, a more efficient shortest path algorithm becomes very important. For example, a police car needs the shortest route through a city to a crime scene. This requires the shortest paths computed quickly and frequently each time it is needed because the traffic conditions may be different every hour. Another good example is communication networks. As the network information traffic conditions or the network connections keep changing, the computer's knowledge of shortest paths of the network will need to be updated very often.

Many efficient algorithms have been developed for different kinds of shortest path problems. The shortest path problem has three main categories, namely the single source shortest path (SSSP) problem, the all pairs short-

est path (APSP) problem and the single pair shortest path (SPSP) problem. Let $G = (V, E)$ be a directed graph where V is the set of vertices in the graph and E is the set of edges. If there is a designated source vertex s, the single source shortest path problem is to find a shortest path from s to every vertex v in G. The all pairs shortest path problem is to find shortest paths from one vertex v to another vertex w for all vertices in G. The single pair shortest path problem is to find the shortest path from one assigned vertex v to another assigned vertex w in G.

For unrestricted graphs, Dijkstra's algorithm [9] is still the most efficient algorithm for the SSSP problem. When Dijkstra's algorithm uses an efficient data structure, such as Fibonacci heaps [10] or 2-3 heaps [24] in its priority queue manipulations, it can achieve $O(m+n\log n)$ time where n is the number of vertices, and m is the number of edges of a given graph. We assume that the SSSP algorithms that appear in this paper use Fibonacci heaps or 2-3 heaps for their priority queue manipulations. However, for restricted digraphs, we have efficient alternatives, like acyclic graphs with $O(m+n)$ time [26] and planar graphs with $O(n\sqrt{\log n})$ time [12].

If a graph is *nearly acyclic*, obviously we should not use the conventional algorithms for general graphs. If we use Dijkstra's algorithm, it does not count the underlying graph structure, and always involves n delete-min operations. In order to efficiently compute the shortest paths for nearly acyclic graphs, several specialized algorithms have been published [1, 3, 23, 20, 22]. These works have shown that we can reduce the number of delete-min operations performed in priority queue manipulations.

However, those specialized algorithms are based on two different measures of what a nearly acyclic graph is. (1) Takaoka gives a definition of acyclicity — the degree of cyclicity of a graph G, $cyc(G)$, is defined by the maximum cardinality of the strongly connected components (sc-components) of G. When the $cyc(G)$ is small, he categorizes the given graph as a nearly acyclic graph [23]. The time complexity of his algorithm using this method is $O(m + n\log k)$ where $k = cyc(G)$ (2) Saunders states that a nearly acyclic graph is a graph that contains relatively few acyclic sub-graphs, each sub-graph of which is dominated by a vertex, called a *trigger* [21]. Obviously, removal of triggers cuts all cycles in the graph. The time complexity of his

algorithm using this method is $O(m + n\log r)$ where r =*number of triggers* Saunders' idea is similar to the measure used by Abuaiadh and Kingston [3], who say a graph is nearly acyclic if there are very few simple cycles in the graph. Note that we need preprocessing to use the above properties of near acyclicity. Here, we measure the near acyclicity of the graph by those parameters such as $k = cyc(G)$ and $r = number\ of\ triggers$. The smaller the values of the parameters are, the more acyclicity the graph has. These two measures (1) and (2) are independent and can not explain one another. We will have a more detailed review of these works in Chapter III.

Therefore, the motivation of this research is to continue to investigate the third parameter of the time complexity. For the original Dijkstra $T(n) = O(n^2)$. After Fibonacci heap was invented, it became $T(m, n) = O(m + n\log n)$. If the graph is sparse, that is, m is small, this complexity is more sensitive to m. When the graph is dense, that is, $m = O(n^2)$, we come to the original complexity, and when sparse, i.e., $m = O(n)$, it becomes linear. Another variation is the case where the maximum integer values of edge costs are limited. Then by a clever method, we have $T(m, n, c) = O(m+n\log c)$. If $c = O(n^k)$ in general, we have $O(m + n\log n)$, the original complexity. If c is small, 2 or 3, it becomes $O(m+n)$, linear. This is a natural generalization of the plain Fibonacci version. Similarly we can introduce parameters $k, r, r', ...$ from the nearly acyclic approach.

In the real world, we look at a workflow graph in a life cycle of a software project development (see Figure 1.2). People in the IT industry basically use an iterative or a waterfall approach to a software development. The presentations of the two approaches are cyclic and acyclic respectively. However, a software development consists of many projects. From a project workflow graph we can find cyclic and acyclic graph structures.

In Figure 1.2, a project manager starts a new project at step 0 based on a marketing plan or a customers' request. The manager highlights the project requirements, and decides which part of their software code will be modified or added. So the manager can assign the project to a developer who is familiar with that area. After receiving the project at step 1, the developer analyzes the project data, and then moves to the step 2 to design user interfaces, methods and classes to meet all the requirements. At step

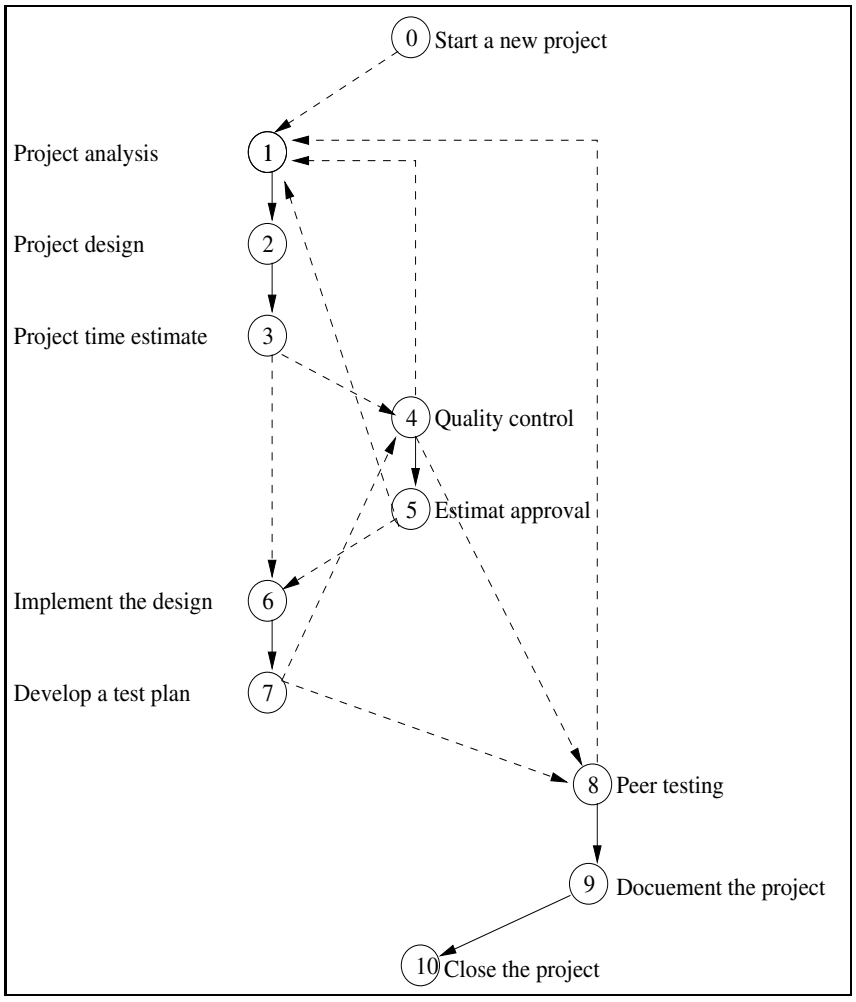

Figure 1.2: A workflow graph

3, the developer estimates the time for implementing his/her own design. When the estimate is done, it goes back to the project manager at step 4 if the estimate goes over a development time threshold. Or, the project goes straight to step 6 if the estimate is under the threshold. If the manager receives a project design document with estimate at step 4, he/she checks the quality of the design. Then at step 5 the manager decides if the estimate is realistic. If everything goes smoothly, the workflow moves to step 6 where the developer implements his/her design. If not, the manager declines the workflow back to step 1. When the developer has implemented the design at step 6, he/she writes up a test plan about the implementation at step 7.

Then, the developer passes that to a tester to do a peer testing at step 8 if the complex of the implementation is under a project complexity threshold. Otherwise, the workflow goes back to step 4 where the project manager needs to look at the code before assigning the project to the tester. If the tester finds any bugs or functional faults at step 8, he/she must decline the workflow back to step 1. If no bugs or functional faults are detected, he/she documents any new or changed functionalities or behaviors in their software at step 9. Then, the project manager closes the project at step 10. The costs of the edges in the graph can be the time to complete certain tasks in the process. For instance, if it takes one hour to analyze the project data at the step 1, the cost of the edge from Node 1 to Node 2 in the graph is 1.

After the project is closed, an escalation officer looks at the outcome of the project, and then carries out the marketing plan or communicates to their customers about the changes in their software introduced by the project. The shortest path theory applies here when a human resource manager needs to know the earliest time when the escalation officer needs to be available to look at the outcome of the project. For instance, the project starts at step 0 on 9AM, and the shortest path is $\{0, 1, 2, 3, 6, 7, 8, 9, 10\}$. If the sum of the cost of the shortest path is 3, the human resource manager can assign some other tasks to the escalation officer before 12PM.

Chapter II

Theoretical Foundations

The concepts of graphs and algorithms introduced in this chapter are the theoretical foundations of this research. Some of the basic concepts and definitions of the graph theories and of algorithm theories will be introduced in the first two sections. Some definitions may not be universal but suit this thesis for computational purposes. Problems studied in this thesis require a systematic traversal or search of graphs. Well designed methods of traversal can efficiently solve problems related to graphs. Some methods of exploring graphs and representations of graphs are illustrated in the third section. Then, Dijkstra's single source shortest path (SSSP) algorithm [9], which was invented in 1959 and provided the foundation for many of today's shortest path algorithms, will be described in the fourth section. At the last section, the concept of 2-3 heaps by Takaoka [24] will be discussed in detail.

2.1 Introducing Graphs

A graph is defined by a set of *vertices* and a set of *edges* that connect these vertices. Let $G = (V, E)$ represent a graph, where V is the set of vertices, and E is the set of edges. Graph 1 in Figure 2.1 is a graph which can be described as $G = (\{1, 2, 3, 4\}, \{e_1, e_2, e_3, e_4, e_5\})$.

The number of vertices in a graph is denoted by $n =\mid V \mid$ and the number of edges by $m =\mid E \mid$. Both n and m are finite in any graphs used in this thesis.

An edge e is specified by two vertices (*end-points*) which it connects. If the end-points of e are vertices v_1 and v_2 then $e = (v_1, v_2)$ or $e = (v_2, v_1)$. Thus, Graph 1 is also defined as:

$$G = (V, E), V = \{1, 2, 3, 4\},\ E = \{(1, 2), (2, 4), (4, 1), (1, 3), (3, 4)\}$$

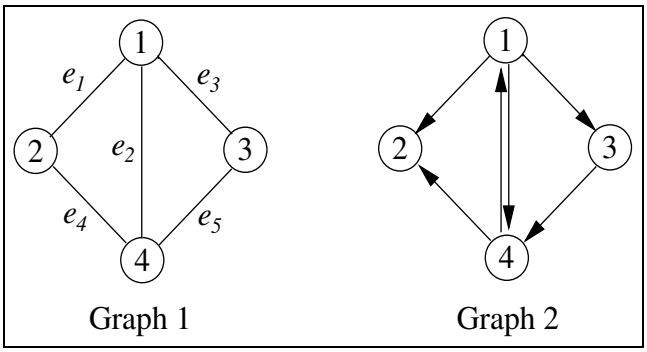

Figure 2.1: An undirected graph and a directed graph

If an edge e has v as an end-point, then we say that e is *incident with* v. If there is an edge $(v, w) \in E$, $v \in V$ and $w \in V$, then v is said to be *adjacent to* w. The *degree* of a vertex v, written $degree(v)$, is the number of edges incident with v [6]. In Graph 1, $degree(1) = 3$, $degree(2) = 2$, $degree(3) = 2$, $degree(4) = 3$. A *sub-graph* of G is a graph obtained by removing some edges and/or vertices from G. The removal of a vertex will remove every edge incident with it. But removal of an edge does not necessarily imply a removal of its end-points unless an end-point has degree 1 [6]. A *path* in a graph is a sequence vertices and edges $\{v_1, e_1, v_2, e_2, ..., e_{i-1}, v_i\}$ such that for $1 \leq j < i$, e_j is incident with v_j and v_{j+1}. When $v_1 = v_i$ and each vertex appear once except the $v_1 = v_i$, the path is a *simple cycle* [6].

In Graph 1, we ignore the directions of the edges. We call this kind of graph an *undirected graph*. In Graph 1, the edge $(1, 4)$ and the edge $(4, 1)$ are representing the same edge. However, if we consider the directions of the edges, a graph with directed edges is called a *directed graph*. Graph 2 in Figure 2.1 shows a *directed graph* where the edge $(1, 4)$ and the edge $(4, 1)$ are different edges. A directed graph is also called a *digraph*. The terms '*digraph*' and *'directed graph'* are interchangeable in this thesis. In this thesis, we focus on solving the shortest paths for directed graphs. Therefore, from now on a graph mentioned indicates a digraph.

If there is a path from a vertex u to another vertex v, we say that v is reachable from u and write as $u \dot{\rightarrow} v$. If $u \dot{\rightarrow} v$ and $v \dot{\rightarrow} u$, we say that u and v are mutually reachable and write as $u \dot{\leftrightarrow} v$. This relation "$\dot{\leftrightarrow}$" is an equivalence relation on the set V. The equivalence classes defined by $\dot{\leftrightarrow}$ are

said to be *strongly connected components* (sc-components) of the digraph. The vertices in the same equivalence class are mutually reachable [28]. For example, the subset $\{1, 3, 4\}$ in Graph 2 is a strongly connected component of the graph. If the whole graph is strongly connected, the graph is a *strongly connected graph* [4].

There are some different families of digraphs. Each family of graphs has some special graph properties that are different to the other graph families. We introduce some of them here. A *planar graph* is a graph which can be drawn on a plane surface with no two edges intersecting. A more precise definition is that a planar graph G is isomorphic to a graph G' such that the vertices and edges of G' are contained in the same plane and such that at most one vertex occupies or at most one edge passes through any point of the plane. G' is said to be embedded in the plane and to be a planar representation of G [6].

An *acyclic graph* is a graph that contains no cycles. Vertices in an acyclic graph can be sorted in a topological order. A *tree* is an undirected acyclic graph where any two vertices of the tree are connected by precisely one path, and a tree with n vertices has $(n - 1)$ edges [6].

A *nearly acyclic graph* is a graph studied in this thesis. Generally speaking, a nearly acyclic graph is close to being acyclic, but nearly acyclic graphs contain cycles. We will have more discussions about this in Chapter III.

An edge can have a *cost*, and we assign a function $cost(v, w)$ to each edge $(v, w) \in E$. In a transport system, the cost of an edge can be the distance between two points. If each edge in a path has a cost, the sum of these costs is the cost of the path. If there is more than one path between two vertices in a graph, then it is possible that one path has a smaller sum of costs than that of another path. This raises three problems of finding shortest paths in a graph, namely *single source shortest path problem*, *all pairs shortest path problem* and *single pair shortest path problem*.

The *single source shortest path problem* is the problem studied in this thesis. This problem is to decide shortest paths from one designated vertex s called *source vertex* to every other vertex in the graph. In any graphs presenting in this thesis, we assume that all vertices are reachable from the *source*. The *all pairs shortest path problem* is to find shortest paths between

all pairs of vertices in the graph. The *single pair shortest path problem* is to compute the shortest path between a *source vertex* and a *destination vertex*. The textbook by Aho, Hopcroft and Ullman [4] and the book by Gibbons [6] provide further introductions to graph theory.

2.2 Introducing Algorithms

In mathematics and computing, an algorithm is a procedure (a finite set of well-defined instructions) for accomplishing some task which, given an initial state, will terminate in a defined end-state [5].

The *complexity* of an algorithm is the number of computational steps that it needs for resolving some input data into a result of the computation [6]. The interest in time efficiency is a concern of the *time-complexity* of algorithms. The concept of *space-complexity* is about the space efficiency of algorithms. The space-complexity is not treated in this thesis, so the term *complexity* refers to *time complexity* and is used in this thesis without ambiguity.

The quantity of the input data is called a *problem size*. For the studies in this thesis, the problem size is determined by one or both of the variables n and m. That is, the number of vertices in a graph and the number of edges in the graph. Time complexity is expressed using the big-O notation. When the number of computational steps is determined by n and m in a function such as $(2m + n\log n + n)$, then the complexity of the algorithm is described by the simplest representative function $O(m + n\log n)$ because the low order terms and constants of a function can be ignored in determining the overall order [6].

Time complexity is a main method of comparing efficiencies of algorithms. For example, let us assume that algorithms Alg_1 and Alg_2 are algorithms for solving the same problem. Alg_1 and Alg_2 have time complexities $O(m)$ and $O(m^2)$ respectively, and their actual running times are described by functions $f_1(m) = 100m$ and $f_2(m) = m^2$ respectively. Although Alg_1 has a larger constant factor (100) associated with its running time than the factor (1) in the running time function of algorithm Alg_2, we say that algorithm Alg_1 is more efficient than Alg_2 because Alg_1 has a lower time complexity

asymptotically. As m increases, the time consumed by Alg_1 grows slower than the time consumed by Alg_2. Thus, Alg_1 is theoretically more efficient than Alg_2 for increasingly large input of m, in this case $m > 100$ [21].

There are three categories of time complexity of algorithms, namely *worst-case time complexity*, *average-case time complexity* and *best-case time complexity*. The worst-case time complexity is the largest amount of time that an algorithm will spend on arbitrary input. The average-case time complexity is also called an *expected time complexity*, which is the average (or expected) running time of an algorithm on arbitrary input. The best-case time complexity is the smallest amount of time that an algorithm will need to compute arbitrary input [21]. The research work described in this thesis is basically concerned with the *worst-case* time complexity analysis of algorithms.

Another aspect to be taken into account is the *computational models*. Shortest path algorithms are designed generally based on two variants of the Random Access Machine (RAM) model. One is the *comparison-addition* model, and all the algorithms reviewed or developed in this thesis use this model. The *comparison-addition model* works with real-valued edge costs and only allows comparison and addition operations on edge weights and numbers derived from them, and each operation is executed within constant time. Another one is called a *word RAM* model. This model works with integers (machine words) of a limited number of bits. Beside comparison and addition operations, it provides some more complicated operations like subtraction, bit shift, and logical bit operations, and each operation is also assumed to consume a constant amount of computing time [21, 16].

2.3 Graph Data Structures

There are two data structures commonly used to represent a graph, namely *adjacency matrices* and *adjacency lists* [14].

An *adjacency matrix* for a digraph $G = (V, E)$ with n vertices is an $n \times n$ matrix M:

$$\begin{aligned} M(i,j) &= 1 \text{ if } (i,j) \in E \\ &= 0 \text{ otherwise} \end{aligned}$$

Obviously, M is generally asymmetric since G is a digraph, and a spec-

ification of M requires $O(n^2)$ computer memory space. This kind of data structure is suitable for a dense graph because M will be fully used to store information of edges. An advantage of this data structure is that it takes $O(1)$ time to check the existence of an edge by simply looking up array entry $M(i, j)$ [14].

An *adjacency list* on the other hand only stores information of edges of a graph. That is, each vertex of the graph has an associated list of its adjacent vertices. Clearly, this data structure requires $O(n+m)$ space. When a graph is sparse, the adjacency list is a much more space efficient data structure for storing the graph than using an adjacency matrix data structure. If an algorithm continuously visits outgoing edges of a vertex but does not randomly access the data of outgoing edges of a vertex, then it takes the same amount of time to visit all the outgoing edges stored in an adjacency matrix or an adjacency list [14].

All the shortest path algorithms introduced in this thesis are intended to solve the SSSP problem in sparse graphs, and they consecutively access the data of outgoing edges of vertices rather than randomly check the existence of edges. Therefore, all the graphs appearing in the rest of this thesis are represented using the adjacency list data structure.

2.4 Exploring Graphs

When we design graph algorithms, we often need a method for exploring the vertices and edges of a given graph. The adjacency lists allow us to pass from a vertex v to one of its adjacent vertices, and thus traverse through the graph. Since there may be more than one neighbor vertices of v, a decision must be made as to which neighbor vertex w should be visited next, and we may need to establish a priority of eligible candidates. *Depth-first search* (DFS) and *breadth-first search* (BFS) have been invented for this purpose [14].

The *depth-first search* and *breadth-first search* both traverse each edge of the graph exactly once in the forward and reverse directions and all vertices are visited. Thus, they are both useful in exploring a graph. However, the choice of which search method to use will often affect the efficiency of

the algorithm [14]. The depth-first search technique is most often used in this thesis, but the a combination of depth-first and breadth-first search introduced by Saunders and Takaoka [20] will also be used by algorithms presented in later chapters.

For the purpose of describing algorithms, we assume that the readers have some experience of computer programming in a high-level language like **C** or **PASCAL**. However, the algorithms in this thesis are written in a simple model language rather than an actual programming language.

2.4.1 Depth-First Search and Breadth-First Search

Depth-First Search:

The basic idea of depth-first search is to start from a vertex v_0, then visit an *unvisited* vertex v_1 which is adjacent to v_0 but not equal to v_0, continuing with an *unvisited* vertex v_2 adjacent to v_1 but not v_0 or v_1, and so forth. As the search goes deeper and deeper into the graph, it will eventually go to a vertex b which has no *unvisited* neighbors. When such a vertex b is visited, the search returns to the vertex a immediately preceding in the search and restart another search from a [14].

Algorithm 2.1 is a depth-first search algorithm [14]. Let a set $OUT(v)$ be a set of vertices reachable from a vertex v by a single edge.

Algorithm 2.1. The Depth-First Search Algorithm

1. **procedure** $DFS(v)$ **{**
2. $visited[v] \leftarrow True$;
3. **for** all $w \in OUT(v)$ **do {**
4. **if** $visited[w] = False$ **then** $DFS(w)$;
5. **}** .
6. **}**

/***** main program *****/

7. **for** all $v \in V$ **do** $visited[v] \leftarrow False$;
8. **for** all $w \in V$ **if** $visited[w] = False$ **do** $DFS(w)$;

In this algorithm, the procedure $DFS(v)$ is a *recursive procedure*, and it calls itself. What happens behind the scene is that when a call to itself is executed, the current values of all variables local to the procedure and the line of the procedure which executed the call are stored at the top of a stack. Then, when control is returned, the computation can continue where it had left off with all the resources recovered. High level programming languages like **C** and **PASCAL** allow recursive subroutines and set up the stack automatically for programmers [14]. Algorithms appearing in this thesis are implemented in **C** language.

Breadth-First Search:

The basic idea of the breadth-first search is to start from a vertex v and put it on an initially empty queue Q of vertices to be visited. Then, the search repeatedly removes the vertex w at the head of Q, and then places onto the queue all vertices adjacent to w that have never been enqueued. Algorithm 2.2 is a breadth-first search algorithm [14].

Algorithm 2.2. Breadth-First Search Algorithm

1. **procedure** $BFS(v)$ **{**
2. **while** $Q \neq \emptyset$ **{**
3. $v \leftarrow$ head of Q;
4. $Q \leftarrow Q - v$;
5. $visited[v] \leftarrow True$;
6. **for** all $w \in OUT(v)$
7. **if** $visited[w] = False$ **and** $w \notin Q$ **then** $Q \leftarrow Q + w$;
8. **}** .
9. **}**

/***** main program *****/

10. **for** all $v \in V$ **do** $visited[v] \leftarrow False$;
11. $Q \leftarrow \emptyset$;
12. **while** $w \in V$ **and** $visited[w] = False$ **do {**
13. $Q \leftarrow Q + w$;
14. $BFS(w)$;
15. **}**

A *restricted-depth-first search* introduced by Saunders and Takaoka in 2003 [20] borrowed this idea in their design, and their algorithm will be reviewed in Chapter III.

2.4.2 Depth-First Search for Strongly Connected Components Algorithm

When do a depth-first search in a directed graph, we define a set of edges, F, where edges can only be added to F if they are directed away from the current vertex being visited. If no such edge exists to an unvisited vertex from those already visited, then the next vertex to be visited becomes the root of an out-tree [6]. For a graph presented in Figure 2.2, Figure 2.3 illustrates such an application of the depth-first search for the graph. The search partitions the edges of the digraph into four types:
(i) a set of spanning-out forest edges, F.
(ii) a set of back-edges, B_1, which are directed from descendants to ancestors.
(iii) a set of forward-edges, B_2, which are directed from ancestors to descendants.
(iv) a set of cross-edges, C, which connect two vertices neither of which is a descendant of the other.

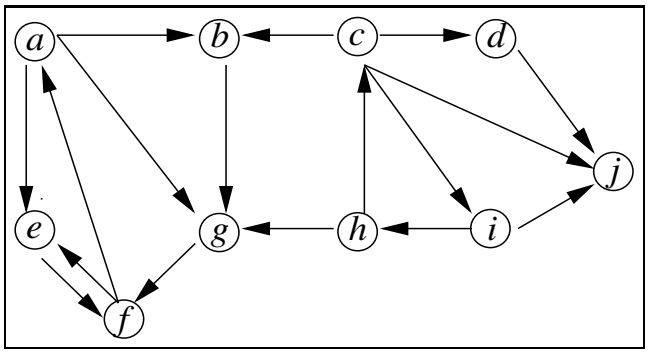

Figure 2.2: A directed graph

Algorithm 2.3 is an algorithm of depth-first search for sc-components by Tarjan [25]. We associate a parameter $lowLink(v)$ with each vertex v. If the vertices are labeled by variable $visitNum(v)$ according to the order in which they are visited in a depth-first search, then $lowLink(v)$ is to be the smallest of $visitNum(v)$ and those vertices which are connected by a back-edge or a

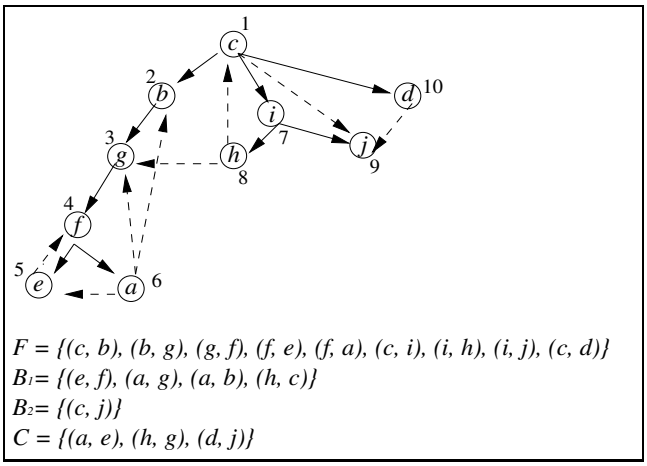

Figure 2.3: A spanning tree of the directed graph in Figure 2.2

cross-edge. Readers can find more theoretical proof from [6, 14, 28].

In Algorithm 2.3, the procedure $CONNECT(v)$ performs the depth-first searches for sc-components. A variable c is a count of the global visit order at line 2. At line 4, all the candidates for sc-components are stored in a first-in-last-out stack, $STACK$. Line 3 initializes $lowLink(v)$ to its maximum possible value and line 8 updates $lowLink(v)$ if a son of v, w, is found such that $lowLink(w) < lowLink(v)$. At line 10 it further updates $lowLink(v)$ if an edge (v, w) in B_1 or C is found such that the root of the sc-component containing w is an ancestor of v. Notice that at line 10 $visitNum(w) = 0$ and so w has been previously visited and since $visitNum[w] < visitNum[v]$ for the update to take place, edge (v, w) cannot be a forward-edge. Also, since w is stacked, the root of the strongly connected component containing w has yet to be identified [6]. Line 11 identifies a root of an sc-component. Those vertices on the stack after the root induce an sc-component. They are then popped out of the stack, including the root vertex [6].

Algorithm 2.3. The Depth-First Search for Strongly Connected Components Algorithm.

1. **procedure** $CONNECT(v)$ **{**
2. $visitNum[v] \leftarrow c$; $c \leftarrow c + 1$;
3. $lowLink[v] \leftarrow visitNum[v]$;
4. $STACK \leftarrow STACK + \{v\}$;
5. **for** all $w \in OUT(v)$ **do**
6. **if** $visitNum[w] = 0$ **then do {**
7. $CONNECT(w)$;
8. $lowLink[v] \leftarrow min\{lowLink[v], lowLink[w]\}$;
9. **}**
10. **else if** $visitNum[w] < visitNum[v]$ **and** $w \in STACK$
then $lowLink[v] \leftarrow min\{lowLink[v], visitNum[w]\}$;
11. **if** $lowLink[v] = visitNum[v]$ **then do {**
12. **repeat**
13. $w \leftarrow POP(STACK)$;
14. **until** $w = v$;
15. **}**
16. **}** .

/***** main program *****/

17. $c \leftarrow 1$; $STACK \leftarrow \emptyset$;
18. **for** all $v \in V$ **do** $visitNum[v] \leftarrow 0$;
19. **for** all $w \in V$ **if** $visitNum[w] = 0$ **do** $CONNECT(w)$;

Figure 2.4 and Figure 2.5 show an application of the depth-first search for sc-components in the graph presented in Figure 2.2 and the spanning tree of the graph in Figure 2.3. In Figure 2.4, the values of $visitNum(v)$ and $lowLink(v)$ are decided during the course of computation. In Figure 2.5 we illustrate the state of the stack just before a vertex is found for which $lowLink(v) = visitNum(v)$ and just after the vertices of a sc-component have been popped from it. We also indicate within which of the recursive calls of CONNECT() the strongly connected components are found.

Vertex	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*
Visit Number	*6*	*2*	*1*	*10*	*5*	*4*	*3*	*8*	*7*	*9*
Low Link	*2*	*2*	*1*	*10*	*4*	*2*	*2*	*1*	*1*	*9*

Figure 2.4: The visit number and low link values of vertices of the directed graph in Figure 2.2

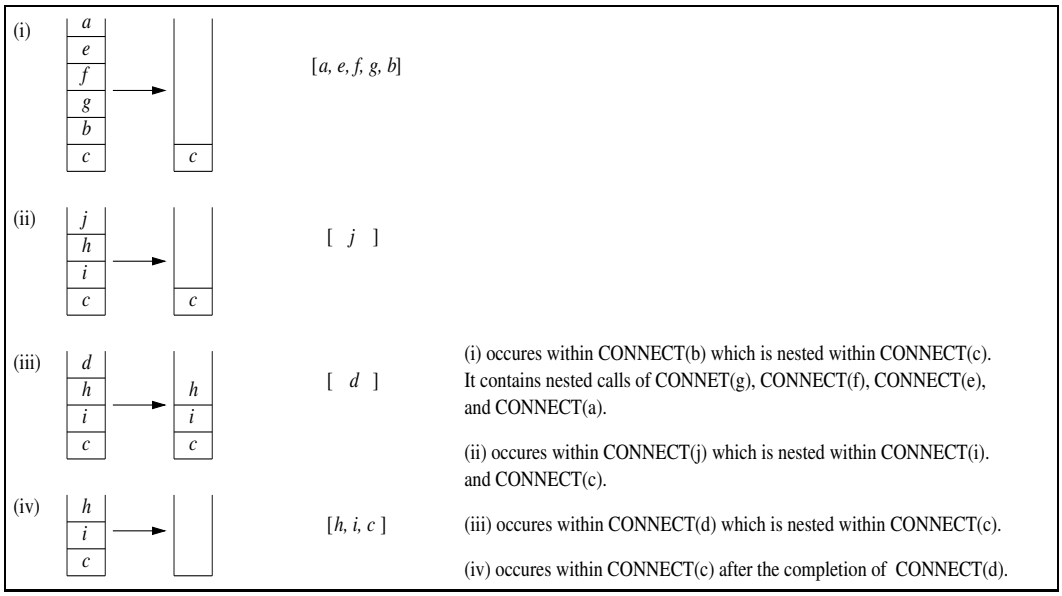

Figure 2.5: The states of the stack

2.5 Dijkstra's Single Source Shortest Path Algorithm

After discussing all the concepts of graphs and algorithms, we now analyze a well-known single source shortest path (SSSP) algorithm invented by Dijkstra in 1959 [9, 18]. Dijkstra's algorithm works for a digraph $G = (V, E)$ with non-negative edge weights, and computes the shortest paths from a source vertex $s \in V$ to all other vertices. The specified vertex s is called a *source*. If we apply Dijkstra's algorithm for every vertex in V, we solve the *all pairs shortest path* (APSP) problem. An alternative to Dijkstra's APSP algorithm was invented by Floyd in 1962 [11], which could efficiently solve the APSP problem on dense graphs. As we said earlier, Dijkstra's algorithm only works for graphs with non-negative edge weights, but the Bellman-Ford's algorithm

(described in [7]) allows graphs to have negative edge weights.

In Dijkstra's algorithm (see Algorithm 2.4), there are three sets, S, F and $V - S - F$, called a *solution set*, *the frontier set* and *the unknown world* respectively [28]. The algorithm maintains a variable $d[v]$ for each vertex $v \in V$, which is the distance of a path to v. When the input data has been precessed, the $d[v]$ should maintain a distance value of the shortest path from the source s to the vertex v. So, the set S is the set of vertices to which the algorithm computed the shortest distances $d[v]$. The set F is a set of vertices which are adjacent with some vertices in set S.

Algorithm 2.4. Dijkstra's SSSP Algorithm

1. **for** all $v \in V$ **do** $d[v]=\infty$;
2. solution set $S = \emptyset$;
3. the source vertex s and $d[s] \leftarrow 0$;
4. $S \leftarrow S \cup \{s\}$;
5. **for** $v \in OUT(s)$ **do** $d[v] \leftarrow cost(s, v)$;
6. frontier set $F = \{v \mid (s, v) \in E\}$;
7. **while** $F \neq \emptyset$ **do {**
8. $v = u$ such that $d[u]$ is minimum among u in F;
9. $F \leftarrow F - v$; // *delete-min*
10. $S \leftarrow S \cup \{v\}$;
11. **for** each $w \in OUT[v]$ **and** $w \notin S$ **do**
12. **if** $w \in F$ **then** $d[w] \leftarrow Min\{d[w], d[v] + cost(v, w)\}$;
13. **else do {**
14. $d[w] \leftarrow d[v] + cost(v, w)$;
15. $F \leftarrow F \cup \{w\}$;
16. **}**
17. **}**

The basic design of this algorithm is that: The distance $d[v]$ for all $v \in F$ is a distance of the *shortest path* that lies in S except for the end-point v. The algorithm finds a vertex v such that $d[v]$ is minimum among vertices in F. Since there should be no other shorter route to v, the vertex v can be included into set S. From the vertex v, it updates the distances $d[w]$ for all

$w \in V - S$ and $(v, w) \in E$ [28]. In Algorithm 2.4, $OUT(v) = \{w \mid (v, w) \in E\}$. $cost(v, w)$ is the cost of edge $(v, w) \in E$.

Apparently, Dijkstra's algorithm can be implemented using simple array data structure, and then the complexity of the algorithm becomes $O(n^2)$. The determination of minimum v at line 8 can be achieved with at most n comparisons without using complex priority queue data structures. Line 11 requires no more than n assignments. Both lines 8 and 11 are contained within the body of the *while* statement beginning at line 7. To find the shortest paths for all vertices, the *while* body needs to be executed $(n - 1)$ times. Thus, the overall time complexity is $O(n^2)$ [18, 30].

In Dijkstra's algorithm, the initial distance values are given by $d_0[s] = 0$ and $d_0[v] = \infty$ for all other $v \in V$. Takaoka (1998) proposed a *generalized single source* (GSS) problem in which there is no source vertex superior to other vertices, but every vertex has an initial distance. In the *generalized single source* algorithm (Algorithm 2.5), all vertices have initial distances from a hypothetical source s. The vertex with the minimum distance is chosen first, and corresponding updates are done, and so forth. The computation is similar to Dijkstra's SSSP algorithm except for the initial distance distribution.

Algorithm 2.5. GSS Algorithm

1. **for** all $v \in V$ **do** $d[v] \leftarrow d_0[v]$;
2. Organize V in a priority queue Q with $d[v]$ as a key;
3. solution set $S = \emptyset$;
4. **while** $S \neq V$ **do {**
5. Find v from Q with minimum key and delete v from Q;
6. $S \leftarrow S \cup \{u\}$;
7. **for** $w \in V - S$ **do {**
8. $d[w] \leftarrow min\{d[w], d[v] + cost(v, w)\}$;
9. Reorganize Q with new $d[w]$;
10. **}**
11. **}**

Obviously, the time complexity of Algorithm 2.5 is the same as that

of Dijkstra's algorithm since the GSS algorithm is a generalized version of Dijkstra's algorithm.

However, we can improve the time complexity of Dijkstra's algorithm and the GSS algorithm using more sophisticated data structures like the 2-3 heap [24] or the Fibonacci heap [26], and the time complexity can be improved to $O(m + n\log n)$. The following section introduces the concepts of the 2-3 heap and the amortized cost analysis of the 2-3 heap.

2.6 2-3 Heap

This section describes the concepts of the 2-3 heap introduced by Takaoka in 2003 [24]. All SSSP algorithms presented in this thesis use 2-3 heaps in their priority queue manipulations. The 2-3 heap is used to improve the asymptotic running time of Dijkstra's algorithm for computing shortest paths in a graph.

Generally speaking, a 2-3 heap is a heap with the nodes' value as the keys, and a node with the smallest value is placed at the top or root of the 2-3 heap. If there are n nodes, we will have maximum $\log n$ 2-3 heaps [24]. For the Dijkstra's SSSP algorithm, we compare maximum $\log n$ times to find the node with the smallest value, so that the Dijkstra's algorithm can achieve a worst-case time complexity $O(m + n\log n)$.

The 2-3 heap (see Figure 2.2) is a data structure that is derived from an analogy with data structure 2-3 tree [28]. The path lengths of a 2-3 heap is bounded by $\log n$. The 2-3 heap links up to three roots of three trees in non-decreasing order according to their key values.

The 2-3 heap data structure uses a concept *trunk* to construct 2-3 heaps. That is, trees in a 2-3 heap are made up of trunks, and trunks are formed by linking nodes that have the same *dimension* in a chain. A node with a smaller key occupies a higher position in a trunk. The 2-3 heap allows one *main trunk* for each *dimension* (see Figure 2.2). In this description, length is defined as the number of nodes in the trunk.

Now, let us look at the cost for each *delete-min*, *decrease-key* and *insertion* operation of the 2-3 heap. In a *delete-min* operation, it takes at most $\lceil \log(n + 1) \rceil$ comparisons to find the minimum. Then, we break apart the

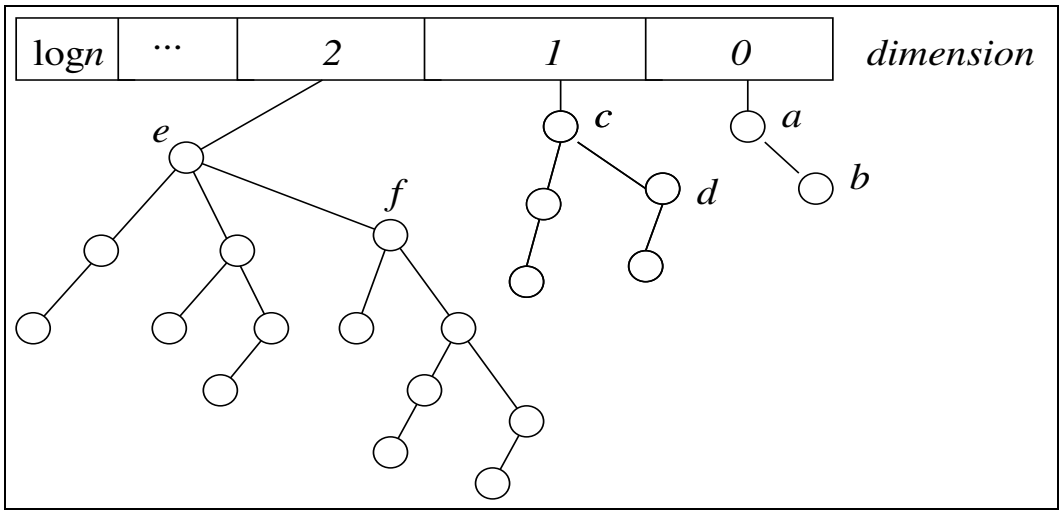

Figure 2.6: Nodes a and b have $dimension = 0$, and they form a main trunk $T(1)$. Nodes c and d have $dimension = 1$, and they form a main trunk $T(2)$. Nodes e and f have $dimension = 2$, and they form a main trunk $T(3)$.

subtree under the root with the minimum, so, the cost for one *delete-min* operation is bounded by $\log(n+1)$, and the actual time is $O(\log n)$ [24].

In a *decrease-key* operation, after decreasing the value of the key, it takes at most 2 comparisons in the worst case or an amortized case in non-critical condition. Thus, the cost for one *decrease-key* is 2, and the actual time is $O(1)$ [24]. In critical condition structural reform propagate with amortized cost 0.

In an *insertion* operation, it takes $O(1)$ time to insert a new node to the 0^{th} term of the top level. But it may cause propagating insertions to all the heaps when all the existing heaps have reached their maximum capacity. During propagation comparisons are saved into potential. The actual time is bounded by $O(\log n)$ and the amortized time is $O(0)$ [24].

If there are n insert, n delete-min and m decrease-key operations, in terms of the number of comparisons, the total costs are bounded by $2m + 3n\log n$. Hence, Dijkstra's SSSP algorithm proves to have $O(m+n\log n)$ time complexity when using this data structure.

Fibonacci heap [10] is another data structure that supports n insert, n delete-min and m decrease-key operations in $O(m + n\log n)$ time. When the key value of a vertex v is decreased, the subtree rooted at v is removed and

linked to another tree at the root level of the heap. Each vertex in the heap is allowed to lose at most one child. If a vertex has to lose another child, that will cause an operation called a *cascading cut*. So, the number of children of any vertex in the heap can not have more than $1.44\log n$ nodes [10, 24]. But, this method will not be discussed in this thesis, and readers are referred to [10, 24] for more detail.

Chapter III

Research Outline

This chapter shows which particular area this research has focused on. The first section of this chapter will describe the research area of this thesis. Then, the second section will give a detailed review of related work that has been done. The last section will discuss what contributions this research can make to the area, and discuss possibilities for improving the existing work.

3.1 Research Area

For unrestricted graphs, Dijkstra's algorithm [9] is still the most efficient algorithm for the single source shortest path (SSSP) problem. When Dijkstra's algorithm uses an efficient data structure, such as Fibonacci heaps [10] or 2-3 heaps [24] in its priority queue manipulations, it can achieve $O(m+n\log n)$ time where n is the number of vertices, and m is the number of edges of a given graph. We assume that the SSSP algorithms that appear in this thesis will all use the 2-3 heap for their priority queue manipulations. However, for restricted digraphs, we have efficient alternatives, like acyclic graphs with $O(m+n)$ time [26] and planar graphs with $O(n\sqrt{\log n})$ time [12].

If a graph is *nearly acyclic*, obviously we should not use the conventional algorithms for general graphs. If we use Dijkstra's algorithm, it does not count the underlying graph structure, and always involves n delete-min operations. In order to efficiently compute the shortest paths for *nearly acyclic* graphs, several specialized algorithms have been published [1, 3, 23, 20, 22]. These works have shown that we can reduce the number of delete-min operations performed in priority queue manipulations.

However, there is argument about what a nearly acyclic graph is. (1) Takaoka [23] gives a definition of acyclicity — the degree of cyclicity of a

graph G, $cyc(G)$, is defined by the maximum cardinality of the strongly connected components (sc-components) of G. When the $cyc(G)$ is small, he categorizes the given graph as a nearly acyclic graph [23]. (2) Saunders states that a nearly acyclic graph is a graph that contains relatively few acyclic sub-graphs, each sub-graph of which is dominated by a vertex, called a *trigger* [21]. Obviously, removal of triggers cuts all cycles in the graph. Saunders' idea is similar to the measure used by Abuaiadh and Kingston [3], who say a graph is nearly acyclic if there are very few simple cycles in the graph. Note that we need preprocessing to use the above properties of near acyclicity. Here, we measure the near acyclicity of the graph by those parameters such as $k = cyc(G)$ and $r = number\ of\ triggers$. The smaller the values of the parameters are, the more acyclicity the graph has. These two measures (1) and (2) are independent and can not explain one another. We will have a more detailed review of these ideas in the next section.

The first part of this thesis describes an extended technique of sc-components decomposition. Here, the term *decomposition* means *graph decomposition*. A *graph decomposition* is to decompose a big graph into smaller sub-graphs by using a decomposing method without changing the structure of the original graph. In this way, we can efficiently run a shortest path search in one sub-graph without searching irrelevant parts of the whole graph. As a result, we can speed up a search of the shortest path in the graph. We call the first approach the *higher-order decomposition*, and it will be presented in Chapter IV. It is developed upon Takaoka's technique of strongly connected component decomposition for nearly acyclic graphs [23], which we have mentioned earlier in this section. In the sc-component decomposition, a graph is decomposed into sc-components. Thus, for computing the shortest paths, we run Dijkstra's algorithm only for each sc-component but not the whole graph. When all the sc-components are small, then we can efficiently solve the SSSP problem. Based on the *higher-order decomposition*, we give a refined definition of acyclicity. That is, the degree of cyclicity $cyc^h(G)$ is the maximum cardinality of the h^{th} *order* strongly connected components of G, where h is the number of times that the graph has been decomposed. The original definition introduced by Takaoka [23] can then be presented as: the degree of cyclicity $cyc(G)$ is the maximum cardinality of the 1^{th} *order*

strongly connected components of G.

In the second part of this thesis we combine the two measures of near acyclicity proposed by Takaoka and Saunders into one, and show how to efficiently solve the SSSP problem in nearly acyclic graphs. For preprocessing we use a *hierarchical depth first search* (HDFS) algorithm to decompose graphs, which will be described in Chapter V. This algorithm does 1-dominator decomposition and decomposition on triggers into sc-components at the same time. In the 1-dominator decomposition, the graph is decomposed into acyclic sub-graphs such that each acyclic sub-graph is dominated by a vertex called a *trigger* and any vertex in the sub-graph can be only reached from the outside through the trigger. We say that the trigger dominates this acyclic sub-graph. We sometimes use "acyclic structure" to indicate "acyclic sub-graph", and they are used interchangeably in this thesis. A 1-dominator set is the set of the acyclic structures. The computing time for the graph preprocessing is $O(m)$. We degenerate the graph in such a way that each new vertex is a trigger in the original graph and a new edge exists from a vertex u to a vertex v if there is an edge from the corresponding acyclic sub-graph dominated by u to v. Let r be the number of triggers in the 1-dominator set and parameter l be the maximum size of sc-components in the degenerated graph. Using this preprocessing, we show that we can efficiently solve the SSSP problem for any kind of nearly acyclic graph in $\mathrm{O}(m+r\log l)$ time.

In the third part of this thesis, we modify the concept of 1-dominator sets to define 1-2-dominator sets, and the generalized concept will be described in Chapter VI. In a 1-2-dominator set, generally speaking, one or two trigger vertices cooperatively dominate an acyclic structure in the graph. Such an acyclic structure is larger than or equal to that in the 1-dominator set. As a result, we will need fewer trigger vertices to cover the whole graph, that is, $r' \leq r$, where r' is the number of triggers in the 1-2-dominator set, and r is the number of trigger vertices in the 1-dominator set. Considering efficient shortest path algorithms only do delete-min operations on trigger vertices, fewer trigger vertices can reduce the time for computing shortest paths. When r' is much smaller than r, we can gain efficiency in computing SSSPs for a nearly acyclic graph in $\mathrm{O}(m + r'\log r')$ time. We present algo-

rithms to achieve $O(m + r^2)$ time to compute the 1-2-dominator set in the graph.

Chapter VII shows experimental comparisons, which demonstrate the practical effectiveness of new algorithms presented in previous chapters and their associated acyclic decomposition performances.

Chapter VIII gives concluding remarks, and discusses what future work can be done in this area.

3.2 Review of Related Work

Abuaiadh and Kingston (1993) suggested that the inherent complexity of the shortest path problem depends on the cycle structures of a graph as well as on its size [1]. They proved that decomposing a graph into parts could speed up the computation of solving the shortest path problem. They gave an algorithm with $O(m+n\log t)$ time complexity where t was the number of delete-min operations needed in the priority queue manipulations. For nearly acyclic graphs, t was expected to be small, so their algorithm could efficiently solve the SSSP problem [2]. However, the value of t is defined by an algorithm, and had no direct relation with the static structure of the graph. So, t was a hypothetical value depending on the algorithm chosen for graph decomposition. Later in 1994, they introduced another algorithm with time complexity $O(m+k\log k)$, where k was the number of cycles in a graph and the graph decomposition used was between *tree decomposition* and *acyclic decomposition* (Saunders 2004). That was improved by Saunders and Takaoka (2005), who defined the concept of 1-dominator set and this would be discussed later in this chapter.

3.2.1 Strongly Connected Components Decomposition

Takaoka (1998) gave a definition of acyclicity. The degree of cyclicity of a graph G, $cyc(G)$, was defined as the maximum cardinality of sc-components of G. When $cyc(G)$ was small, he defined the given digraph G to be nearly acyclic. When $cyc(G) = k$, he gave an algorithm with $O(m+n\log k)$ time complexity. Take Figure 3.1 for example, the degree of cyclicity of the graph is 3, so that $k = 3$.

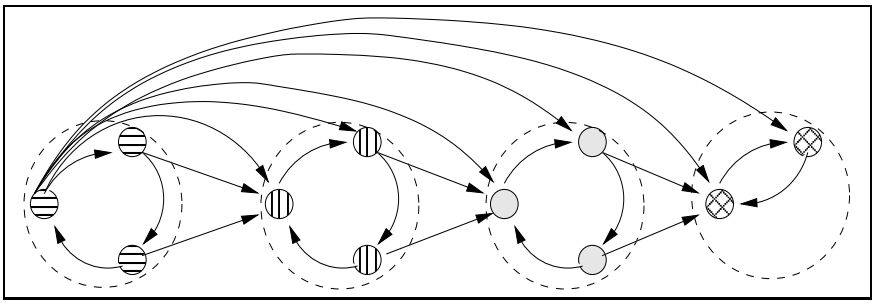

Figure 3.1: Vertex groups with different patterns form four sc-components in a graph. The largest sc-component has three vertices.

The basic idea of this approach lies on the strongly connected components decomposition of a graph. First of all, the Algorithm 2.3 is used to compute *strongly connected components* (sc-components) $V_r, V_{r-1}, ..., V_1$ in a topological order where for $i > j$ there is no edges from V_j to V_i. Now, the graph can be degenerated into an acyclic graph $\widetilde{G} = (\widetilde{V}, \widetilde{E})$ where $\widetilde{V} = \{V_r, V_{r-1}, ..., V_1\}$ and $\widetilde{E} = \{(v, w) \mid (v, w) \in E, v \in V_i, w \in V_j, 1 \leq i, j \leq r \text{ and } i \neq j\}$. For example, the Figure 3.2 is such a degenerated graph of Figure 3.1. Note that $\widetilde{G}$ is an acyclic graph.

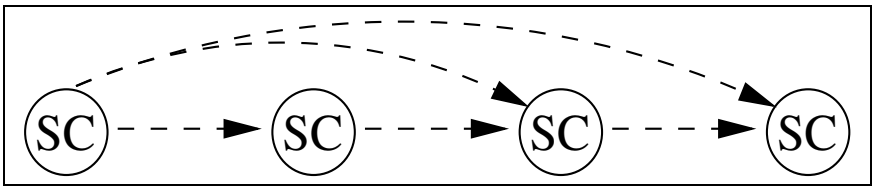

Figure 3.2: A degenerated acyclic graph $\widetilde{G}$ of the graph in Figure 1. $\widetilde{G}$ consists of sc-components and pseudo-edges connecting sc-components.

If we see each sc-component as a sub-graph, then such a sub-graph is defined as $G_i = (V_i, E_i)$ where $V_i \in \widetilde{V}$ and $E_i = \{(v, w) \mid v \in V_i \textit{ and } w \in V_i\}$. The SSSP problem from a source to all other vertices can be solved along the degenerated graph in the topological order from G_r to G_1. For each sub-graph G_i, we solve the *generalized single source* (GSS) problem [23] at line 5 (see Algorithm 2.5 in Chapter II).

Algorithm 3.1 is an algorithm based on the sc-component decomposition. In Algorithm 3.1, (V_i, V_j) indicates a pseudo-edge in degenerated graph $\widetilde{G}$

such that one end-point of the edge is in V_i and the other end-point is in V_j. Variable $d[v]$ maintains a distance of a path to vertex v.

In order to analyze Algorithm 3.1, we need the following lemma and proof established by Takaoka [23].

Lemma 1. *Let non-negative integer variables $x_1, x_2, ..., x_n$ satisfy the conditions $(1)x_1 + x_2 + ... + x_n = x$ and $(2)x_i \leq k (i = 1, 2, ..., n)$ for constant integers k and x such that $0 \leq k \leq x$ and $x \leq kn$. Also, let the maximum of the objective function $\sum_{i=1}^{n} f(x_i)$ be denoted by $\phi_n(x)$ where f(x) is such that $f(x) = xg(x)$ and g(x) is a monotone non-decreasing function. Then we have:*

$$\phi_n(x) \leq ng(k)$$

Proof.

$$\phi_n(x) = \sum_{i=1}^{n} k_i g(x_i) \leq \sum_{i=1}^{n} k_i g(k) = ng(k)$$

The value of $\phi_n(x)$ is obtained by setting as many $x_i's$ as possible to k. $\triangle$

Algorithm 3.1. Solve the SSSP problem using sc-components decomposition

1. Compute sc-components $V_r, V_{r-1}, ..., V_1$
2. **for** $v \in V$ **do** $d[v] \leftarrow \infty$;
3. $d[s] \leftarrow 0$; //For source s let $s \in V_r$ without loss of generality
4. **for** $i \leftarrow r$ *to* 1 **do {**
5. Solve the GSS for G_i;
6. **for** V_j such that $(V_i, V_j) \in \widetilde{E}$ **do**
7. **for** $v \in V_i$ **and** $w \in V_j$ such that $(v, w) \in E$ **do**
8. $d[w] \leftarrow min\{d[w], d[v] + cost(v, w)\}$;
9. **}**

In Algorithm 3.1, at line 1, it takes $O(m + n)$ time to compute sc-components. For graphs used in this thesis, we assume that $n \leq m$. So, the complexity of line 1 is $O(m)$. At line 5, the complexity of computing the shortest paths for a sub-graph G_i is $O(m_i + k_i \log k_i)$ where $m_i =\mid E_i \mid$ and

$k_i =\mid V_i \mid$. Then the overall complexity becomes $O(m + \sum_{i=1}^{r}(m_i + k_i \log k_i))$. Now we apply the $k_i \log k_i$ to the $f(x)$ in Lemma 1, we will have the complexity given by $O(m + n \log k)$ where $k = cyc(G)$.

3.2.2 Acyclic Decomposition

Saunders and Takaoka (2005) offered an acyclic decomposition approach for solving the shortest path problems [22]. In this approach, a graph is decomposed into acyclic structures in O(m) time. Each structure is dominated by a trigger vertex. A 1-dominator set is a set of acyclic structures. Note that triggers and acyclic structures correspond one to one. If a nearly acyclic graph has r trigger vertices, they introduce an algorithm with O($m+r\log r$) time complexity [22] for solving the SSSP problem. The new parameter r represents the number of acyclic structures in the graph. Intuitively speaking, an acyclic structure in the 1-dominator set is an acyclic sub-graph such that any vertex inside can be reached from outside only through the associated trigger vertex. We say that the trigger dominates this acyclic structure. As the triggers and acyclic structures correspond one-to-one, we sometimes use the two concepts interchangeably. We also use "acyclic sub-graph" and "acyclic structures" interchangeably. In Figure 3.3 for an example, there are three acyclic structures.

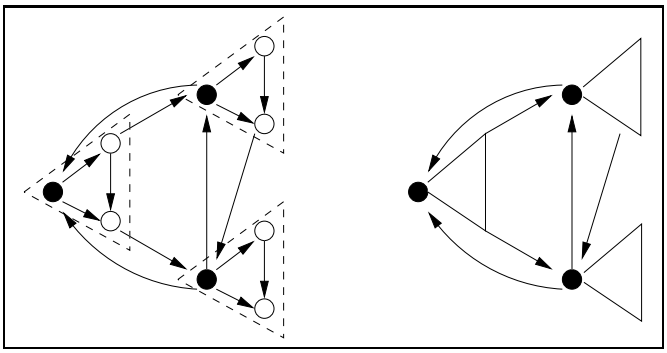

Figure 3.3: The acyclic structures in the left picture are presented in the right as combinations of a node and a triangle where the node represents a trigger vertex and the triangle represents non-trigger vertices dominated by the trigger vertex. There are three trigger vertices in the graph.

Algorithm 3.2 is a 1-dominator decomposition algorithm introduced by

Saunders and Takaoka. The procedure $rdfs()$ stands for restricted depth-first search. It maintains variable *inCount* for each vertex v, which initially contains the total number of incoming edges of a vertex, $|IN(v)|$. When it visits a new vertex, it decreases the *inCount* of the vertex by 1. If *inCount* becomes 0, we say it unlocks the vertex and the search goes forward (line 8), that is, the vertex is traversed. Unlocked vertices are included into a set A for the acyclic structure dominated by the initial vertex w (line 4). At the end of the search from w, A is the acyclic structure dominated by w, and L is the set of vertices tentatively examined (line 6). In this sense, we say, L is the set of seen vertices for possible inclusion into the acyclic structure. At the end $A[v]$ is the acyclic structure dominated by v, and $B[v]$ is the associated boundary set. The algorithm maintains $AC[v]$ which refers to an acyclic structure dominated by trigger vertex v (line 25), and $BS[v]$, which refers to the boundary set of the acyclic structure (line 26). The algorithm also maintains a queue Q containing boundary vertices as trigger vertex candidates (line 30). The algorithm starts from a vertex s, and searches with $rdfs()$ as much as possible. Then it starts searches from boundary vertices again. We assume $n \leq m$ and only treat strongly connected graphs for simplicity. Generalization to a general graph is straightforward. As the graph is strongly connected, the search will eventually come back to s. We note that the searched part from s is only traversed twice. Thus the time complexity of Algorithm 3.2 becomes $O(m)$ proved by Saunders and Takaoka [22].

Algorithm 3.2. 1-dominator decomposition

```
1.  function AcyclicSet(w){
2.    VertexSet A, L, B;
3.    procedure rdfs(u) {
4.      A ← A ∪ {u};
5.      for all v ∈ OUT(u) do {
6.        if v∉L then L←L + {v};
7.        inCount[v]←inCount[v] − 1;
8.        if inCount[v] = 0 // v unlocked
            then rdfs(v);
9.      }
```

```
10.   }
11.   A←∅; L←{w};
12.   inCount[w]←inCount[w] + 1;
        // prevents re-traversal of w
13.   rdfs(w);
14.   VertexSet B ← L − A; // boundary vertices
15.   for all v∈L do inCount[v]←|IN(v)|;
16.   return (A, B);
17. }
/***** main program *****/
18. for all v ∈ V do vertexType[v] ← unknown;
19. for all v ∈ V do inCount[v] ←|IN(v)|;
20. Q ← {s};
21. while Q ≠ ∅ do {
22.   Remove the next vertex u from Q;
23.   if vertexType[u] = unknown then {
24.     (A, B) ← AcyclicSet(u);
25.     Let AC[u] refer to A;
26.     Let BS[u] refer to B;
27.     vertexType[u] ← trigger;
28.     for all v ∈ A do
          vertexType[v] ← non-trigger;
29.     for all v ∈ B do
30.       if vertexType[v] = unknown and v ∉ Q
            then Add v to Q;
31.   }
32. }
```

Algorithm 3.3 is an SSSP algorithm based on the results of acyclic decompositions of graphs. It modifies a general SSSP algorithm introduced by Saunders and Takaoka [20]. Obviously only trigger vertices will be added into the frontier set F (line 8). That means the delete-min operations will not be required by the non-trigger vertices. The distance values $d[v]$ of non-trigger vertices in an acyclic sub-graph will be finalized straightaway with

the *decreaseKey* operation in topological order after their associated trigger vertices reach the minimum values.

Algorithm 3.3. SSSP Algorithm Using Acyclic Decomposition

1. **procedure** *decreaseKey*(u) **{**
2. **for** each $v \in AC[u]$ in topological order **do**
3. **for** each $w \in OUT[v]$ **and** $w \notin S$ **do**
4. $d[w]$=Min$\{d[w], d[v]+cost(v, w)\}$;
5. **}**

/***** main program *****/

6. **for** all $v \in V$ **do** $d[v]=\infty$;
7. solution set $S = \emptyset$;
8. insert all triggers into frontier set F;
9. Let source vertex be s and $d[s] \leftarrow 0$;
10. **if** s is not a trigger **then** *decreaseKey*(s);
11. **while** $F \neq \emptyset$ **do {**
12. $d[u]$ = Min$\{d[u] \mid$ all $u \in F\}$;
13. $F \leftarrow F - u$; // *delete-min*
14. $S \leftarrow S + u$;
15. *decreaseKey*(u);
16. **}**

In line 8, a trigger means a 1-dominator trigger. In Chapter V 1-2-Dominator Sets, this will be a 1-2-dominator trigger.

3.3 Possible Improvements to Existing Approaches

The first research result expected is the generalized technique of the sc-component decomposition. Hence, we may come up with a more general definition of acyclicity, and complete the concept of degree of cyclicity in the sc-component decomposition. We propose a decomposition method called a *higher-order decomposition*. In this approach, a refined degree of cyclicity, denoted by $cyc^h(G)$, is defined to be the maximum cardinality of the h^{th} order strongly connected components of a given graph G, where h is the number of times that the graph has been decomposed. We investigate scenarios where

h is a small constant and we only introduce the first order ($h = 1$) and the second order ($h = 2$) decompositions. According to the new theory, Takaoka's definition can be restated: the degree of cyclicity is the maximum cardinality of the first order strongly connected components of a graph.

The another observation of this research is the difference of definitions of the nearly acyclic graph. Specialized shortest path algorithms reviewed in this chapter improve upon the time complexity of Dijkstra's algorithm when they are applied to different kinds of nearly acyclic graphs. That is, Takaoka's algorithm can efficiently solve the SSSP problem in graphs with $cyc(G) = \mathrm{O}(1)$, but may not be efficient for a graph with a few simple cycles and $cyc(G) = \mathrm{O}(n)$, which is made more efficient by Saunders and Takaoka's algorithm or Abuaiadh and Kingston's algorithm, and vice versa. Thus, there is room to improve upon or complement existing shortest path algorithms for nearly acyclic graphs by introducing new algorithms that promise to solve the SSSP problem for any kind of nearly acyclic digraphs. A new approach called a *hierarchical approach* is such a solution based on the modification to the existing decomposition methods.

Another research interest presented in this thesis is to generalize the 1-dominator set theory to a *multi*-dominator set. Saunders (2004) studied this topic using the same 1-dominator decomposition framework. From now on, we only use a 2-dominator acyclic structure as a demonstration of the multi-dominator acyclic structure. In the 2-dominator acyclic structure, two triggers cooperatively dominate an acyclic structure.

However, we will show that the framework of the 1-dominator set is not efficient for designing the *multi*-dominator set. Work done by Saunders (2004) proved that computing the *multi*-dominator set was too costly in terms of the time consumed when he used the same design framework as that used in the 1-dominator set and the complexity was $O(mr^4)$, where r is the size of the 1-dominator set. Therefore, we present a new framework, a mixture 1-2-dominator set. The 1-2-dominator set will still identify all the possible acyclic structures that can be dominated by two triggers among with the acyclic structures dominated by one trigger. Let r' be the number of triggers in the 1-2-dominator set. When r' is much smaller than r, we can gain efficiency in computing SSSPs for a nearly acyclic graph in $O(m + r'\log r')$

time. We present algorithms to achieve $O(m + r^2)$ time to compute the 1-2-dominator set.

Chapter IV

Higher-Order Approach

This chapter gives a generalized definition of acyclicity based on the concept originally defined by Takaoka [23]. Takaoka's definition of acyclicity is the degree of cyclicity of a graph G, $cyc(G)$. That is defined as the maximum cardinality of the strongly connected components (sc-components) of G. In the generalized definition of acyclicity, another degree of cyclicity, denoted by $cyc^h(G)$, is defined to be the maximum cardinality of the h^{th} order strongly connected components of a given graph G, where h is the depth of the graph decomposed, and we only introduce the first order ($h = 1$) and the second order ($h = 2$) decompositions. According to the new theory, Takaoka's definition can be restated in such a way that the degree of cyclicity is the maximum cardinality of the first order strongly connected components of a graph. Obviously, the preprocessing time is $O(hm)$ and the complexity of solving SSSP problem is $O(hm + n\log\rho)$ where $\rho = cyc^h(G)$ and constant h emphasizes how deeply the graph is decomposed. When $cyc^h(G)$ is a smaller constant, we can efficiently solve the SSSP problem in $O(m+n\log\rho)$ time for fixed h.

4.1 Higher-Order Decomposition

In Takaoka's approach described in Chapter III, the sc-components originally computed in line 1 of Algorithm 3.1 remain fixed through all GSS's. When we start one GSS, there is an opportunity that the corresponding sc-component can further be decomposed.

The basic idea behind *higher-order decomposition* is to remove the first vertex selected in GSS (line 5 of Algorithm 3.1) from the input graph, and then run the sc-component decomposition on the new strongly connected sub-

graphs. The original sc-component decomposition by Takaoka [23] is called *the first order* decomposition, and the sc-component decomposition on the sub-graphs is called *the second order* decomposition, one for sub-graphs is called *the third order* decomposition, and so forth.

After all distance updates are done in lines 6-8 of Algorithm 3.1, we solve the GSS for G_{i-1}. We call the first vertex whose distance is finalized the *pseudo source* of G_{i-1}.

When all the shortest paths have been computed, we find the largest sc-component in the decomposed graph. Let us indicate the size of the largest sc-component by ρ. An SSSP algorithm based on this approach has the worst case time complexity $O(hm+n\log\rho)$. It can be efficient for some nearly acyclic graphs like a graph in Figure 4.3. ρ is dynamically determined by the algorithm.

Thus, we give another definition that the degree of cyclicity $cyc^h(G)$ is the maximum cardinality of the strongly connected components of the decomposed graph G after the h^{th} order decomposition is made.

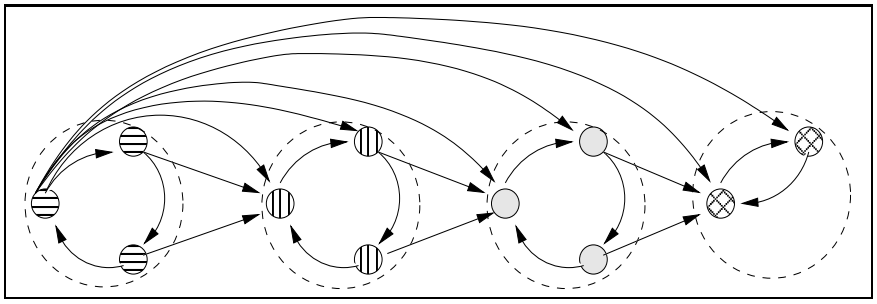

Figure 4.1: Vertex groups with different patterns form four sc-components in a graph. The largest sc-component has three vertices.

Let $G = (V, E)$, $V = \{v_1, v_2, ..., v_n\}$ and $E \subseteq V \times V$. We calculate the sc-components $V_r, V_{r-1}, ..., V_1$. According to the new definition, we call sc-components *the first order sc-components* (see Figure 4.1), and they are defined as $V_r^1, V_{r-1}^1, ..., V_1^1$. Then, we define the *first order degenerated graph* $\widetilde{G}^1 = (\widetilde{V}^1, \widetilde{E}^1)$ where $\widetilde{V}^1 = \{V_r^1, V_{r-1}^1, ..., V_1^1\}$, $\widetilde{E}^1 = \{(v \rightarrow w) \mid edge(v \rightarrow w) \subseteq E, v \in V_i^1 \text{ and } w \in V_j^1, i \neq j\}$ (see Figure 4.2).

We call edges in $\widetilde{E}_1$ *transient edges* since they connect between sc-components. In contrast, we call an edge $(v \rightarrow w)$ an *inside edge* if both v and w belong

to the same sc-component.

Thus, graph $\widetilde{G}^1$ is a decomposed acyclic graph of G [1]. Let us define each sc-components in $\widetilde{G}^1 = (\widetilde{V}^1, \widetilde{E}^1)$ to be sub-graphs. The sub-graphs are defined as $G_i^1 = (V_i^1, E_i^1)$ where $V_i^1 \in \{V_r^1, V_{r-1}^1, ..., V_1^1\}$, $E_i^1 = \{(v \rightarrow w) \mid v \in V_i^1 \ and \ w \in V_i^1\}$ (see SC nodes in Figure 4.2).

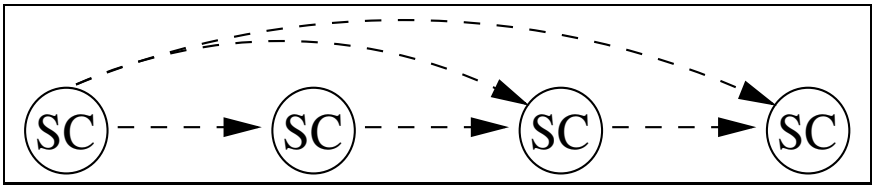

Figure 4.2: A degenerated acyclic graph $\widetilde{G}$ of the graph in Figure 1. $\widetilde{G}$ consists of sc-components and pseudo-edges connecting sc-components.

In the Algorithm 4.1, $OUT(v) = \{w \mid (v \rightarrow w) \in E\}$. $SV = \{$selected vertices$\}$. The purpose of SV is that we obtain sc-components of the graph induced by the set of vertices, $V - SV$. For the first order decomposition $SV = \{s\}$. For each $w \in V$, a variable $Graph[w]$ maintains the index of a sub-graph $\widetilde{G}_i^1$ which w belongs to, and another variable $root[w]$ contains a candidate vertex for the root of the sc-component that w will be part of. The variable $Num[w]$ tells the visit order of w.

We start the $SDFS$ search from calling procedure $SDFS(G, SV)$ at line 17. At line 21 we initialize the vertices in SV to be *visited* because they are selected not to join the sc-components search. At line 22, we call the recursive procedure $CONNECT(v)$ until every vertex except vertices in SV becomes *visited*. From line 23 to line 26, we do $V_r = SV$ and $\widetilde{E}^h = \widetilde{E}^h \cup \{(v \rightarrow w) \mid v \in SV$ and $w \in V\}$ for selected vertices.

Now we look at the recursive procedure $CONNECT(v)$ (line 1). From line 6 to line 7, when the procedure is processing vertex v, $root[v]$ contains a candidate vertex for the root of the component containing v. Initially v itself is the root candidate. When it processes the outgoing edges, new root candidates are updated from those connected vertices which belong to the same component as v.

At line 8, if w is already in a sc-component, we reserve this edge in $\widetilde{E}^h$, and erase this edge from $OUT(v)$. The reason of erasing edges from $OUT(v)$

is that we do not visit these edges again if we have to further decompose the sub-graph which v belongs to. When $CONNECT(v)$ has processed all the outgoing edges of v, $root[v] = Num[v]$ if and only if v is the root of the sc-component containing v (line 10). If line 10 is satisfied, we take all vertices belong to that sc-component out of the STACK, and put them into a sub-graph G_r (line 11 to 15).

Algorithm 4.1. Selective Depth-First Search algorithm (SDFS). In the first order decomposition, $SV = \{s\}$ and $h = 1$.

1. **procedure** CONNECT(v)
2. **{**
3. $visited[v] \leftarrow True$; $Num[v] \leftarrow cnt$; $cnt \leftarrow cnt + 1$;
   $lowLink[v] \leftarrow Num[v]$; $Graph[v] \leftarrow undefined$;
4. $Push(v, STACK)$;
5. **for** each $w \in OUT(v)$ **{**
6. **if** $visited[w] = False$ **then {**
   $CONNECT(w)$; $lowLink[v] \leftarrow min\{lowLink[v], lowLink[w]\}$;
   }
7. **else if** $Num[w] < Num[v]$ **and** $Graph[w] = undefined$
   then $lowLink[v] \leftarrow min\{lowLink[v], Num[w]\}$;
8. **if** $1 \leq Graph[w] < r$ **then**
   $\widetilde{E}_i^h \leftarrow \widetilde{E}_i^h + (v \rightarrow w)$;//transient edge
9. **}**
10. **if** $lowLink[v] = Num[v]$ **{**
11. **Repeat**
12. $w \leftarrow POP(STACK)$; $Push(w, V_r^h)$; $Graph[w] \leftarrow r$;
13. **Until** $w = v$;
14. $r \leftarrow r + 1$;
15. **}**
16. **}**

/***** main function *****/

17. **function** $SDFS(G_i^h, SV)$ //graph G_i^h to be decomposed with source in set SV.
18. **{**

19. $r \leftarrow 1$; $cnt \leftarrow 1$; $STACK \leftarrow \emptyset$; //The r, cnt and $STACK$ are global variables

20. **for** each $v \in V$ **do {**
$visited[v] \leftarrow False$; $Graph[v] \leftarrow NULL$;
}

21. **for** $v \in SV$ **do** $visited[v] \leftarrow True$; //removal of SV from part of the sc-decomposition

22. **for** $w \in V$ **and** $visited[w] = False$ **do** $CONNECT(w)$;

23. $V_r^h \leftarrow SV$;

24. **for** $v \in SV$ **do {**

25. **for** $w \in OUT(v)$ **do**
if $1 \leq Graph[w] < r$ **then** $\widetilde{E}_i^h \leftarrow \widetilde{E}_i^h \cup (v \rightarrow w)$;

26. **}**

27.**}**

If we use SDFS to compute sc-components $V_r^1, V_{r-1}^1, ..., V_1^1$ for a graph G , then the set $\{V_r^1, ..., V_1^1\}$ is sorted from V_r^1 to V_1^1 in a topological order. Here, V_1^1 is the first sc-component computed. V_r^1 is the last one, and $V_r^1 = \{s\}$.

Algorithm 4.2. GSS SSSP algorithm using higher-order decomposition

1.**for** $v \in V$ **do** $d[v] \leftarrow \infty$;

2. $d[s] \leftarrow 0$; $SV = \{s\}$;

3. Call $SDFS(G, SV)$ to compute the first order sc-components $V_r^1, V_{r-1}^1, ..., V_1^1$;

4. **for** $i \leftarrow r$ to 1 **do {**

5. Solve the GSS for G_i^1;

6. **for** V_j^1 such that $(V_i^1, V_j^1) \in \widetilde{E}^1$ **do**

7. **for** $v \in V_i^1$ **do for** $w \in V_j^1$ **do**

8. $d[w] \leftarrow min\{d[w], d[v] + cost(v, w)\}$;

9. **}**

We now use the first order decomposition to solve the SSSP problem (see Algorithm 4.2). In this algorithm, the variable $d[v]$ contains the distance between the source vertex s and v. Initially, it has a maximum value, presented as ∞. At line 3, we decompose the given graph. At line 5 we solve

the generalized single source problem (GSS) for vertices in sub-graph G_i^1 [1]. At line 6, (V_i^1, V_j^1) is a pseudo edge, and $(V_i^1, V_j^1) \in \widetilde{E}^1$ means that there are outgoing edges from G_i^1 to other sub-graphs G_j^1. We update the distance of connected vertices in G_j^1 (see line 8). This algorithm is an generalized version of the GSS algorithm (Algorithm 2.5) presented by Takaoka [23].

Note that Algorithm 4.2 is almost identical to Algorithm3.1 except for superscripts of 1 at some variables.

4.2 Second-Order Decomposition

The second order decomposition is based on the following observation.

Observation 1 *Let us compute sc-components of a graph by the SDFS. After computing all the shortest paths for every vertex in* $V_r^1, V_{r-1}^1, ..., V_{i+1}^1$, *we find a vertex* v *in* V_i^1 *that* $d[v] = min\{d[u] \mid u \in V_i^1\}$. *Then* $d[v]$ *is not updated by any vertex in* $V_i^1, V_{i-1}^1, ..., V_1^1$.

Proof. Because $V_r^1, ..., V_i^1, V_j^1, ..., V_1^1$ is sorted from V_r^1 to V_1^1 in topological order, edges from one sc-component V_i^1 to another one, V_j^1 , must satisfy $i > j$. After calculating all shortest paths for vertices in $V_r^1, V_{r-1}^1, ..., V_{i+1}^1$, we find a vertex v which is in V_i^1 that $d[v] = min\{d[u] \mid u \in V_i^1\}$. Obviously, the $d[v]$ cannot be updated to a smaller value from other vertices in V_i^1 because it is already the smallest one in V_i^1. The $d[v]$ will not be updated from vertices in $V_{i-1}^1, ..., V_1^1$ because there is no edge from them to V_i^1.

Let $\widetilde{G}^1 = (\widetilde{V}^1, \widetilde{E}^1)$ where $\widetilde{V}^1 = \{V_r^1, V_{r-1}^1, ..., V_1^1\}$, $\widetilde{E}^1 = \{(v \rightarrow w) \mid edge(v \rightarrow w) \in E, v \in V_i^1$ and $w \in V_j^1, i \neq j\}$. Let $V_r^1, V_{r-1}^1, ..., V_1^1$ be the first order sc-components of a given graph G where $V_r^1 = \{s\}$. Let us define sub-graphs $G_i^1 = (V_i^1, E_i^1)$ where $V_i^1 \in \{V_r^1, V_{r-1}^1, ..., V_1^1\}$, $E_i^1 = \{(v \rightarrow w) \mid v \mid V_i^1 and w \in V_i^1\}$. For an arbitrary sub-graph G_i^1 let us choose a vertex v such that $d[v] = min\{d[u] \mid u \in V_i^1\}$. Note that $d[v]$ may be updated by different paths from $G_r^1, G_{r-1}^1, ..., G_{i-1}^1$. Let such a vertex v be called *pseudo-source* of G_i^1.

We know that G_i^1 is strongly connected. When we erase all incoming edges of the *pseudo-source* of G_i^1, immediately G_i^1 becomes not strongly con-

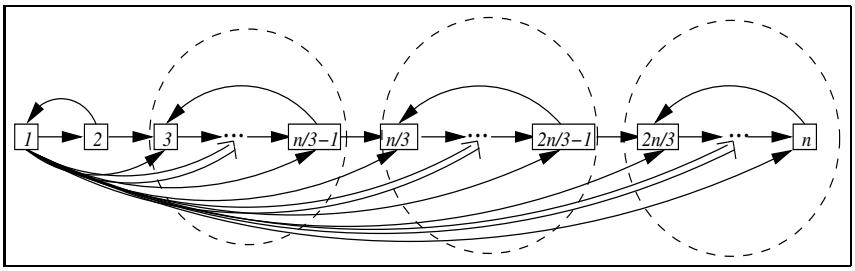

Figure 4.3: Arrow "⇒" indicates *Vertex* 1 has edges to every node unstated. Let Vertex 1 be the source, $\mathrm{cyc}^1(\mathrm{G}) = \mathrm{n}/3$, $\mathrm{cyc}^2(\mathrm{G}) = 1$.

nected. Based on this property, we use the $SDFS$ algorithm to compute sc-components $V_l^2, V_{l-1}^2, ..., V_1^2$ for G_i^1, and $V_l^2 = \{pseudo\text{-}source\}$ where indexing by i is omitted. Let us call sc-components $V_l^2, V_{l-1}^2, ..., V_1^2$ of G_i^1 as the *second order sc-components* of G_i^1. We call the computation of second order sc-components as *dynamic graph decomposition* because the pseudo source vertices are dynamically chosen before the decomposition. The method of *second order sc-components* will be effective for the graph in Figure 4.3.

The Algorithm 4.3 is the algorithm based on dynamic graph decomposition. It starts a recursive call $Dynamic(G, SV)$ at line 16, and $SV = \{s\}$ because it always starts from the first order decomposition. Now let us look at the $Dynamic(G, SV)$. At line 2 we call $SDFS(G, SV)$ to find the h^{th} order sc-components $V_r^h, V_{r-1}^h, ..., V_1^h$. Note that the r will be different from one decomposition to another. From line 4 to line 6, we update $d[w]$ according to edges $(v \rightarrow w)$, $v \in V_i^h$ and $w \in V_j^h, (r \geq i > j)$. At line 7 we find its pseudo source vertex for G_{i-1}^h, and then update set SV with it. At line 9, we terminate the recursive call by not satisfying conditions $(\mid G_{i-1}^h \mid > c_1)$ and $(h \leq c_2)$, where c_1, c_2 are constants.

Case 1. $c_1 = 0$, the procedure will decompose the graph until each V_i^h has only one vertex.

Case 2. $c_1 \geq 1$, each V_i^h will only have a pseudo source and several general vertex.

The constant c_2 is the degree of decomposition. Generally speaking, the

larger c_2 is, the more sc-components we will have. After terminating the recursive call, line 10 solves the SSSP problem for the sub-graph G_i^h.

Algorithm 4.3. For the second order decomposition, $c_2 = 2$

1. **procedure** Dynamic(G, SV) **{**
2. Call $SDFS(G, SV)$ to compute h order sc-components $V_r^h, V_{r-1}^h, ..., V_1^h$;
3. **for** $i \leftarrow r$ to 1 **do {**
4. **for** V_j^h such that $(V_i^h, V_j^h) \in \widetilde{E}^h$ **do**
5. **for** $v \in V_i^h$ **do for** $w \in V_j^h$ **do**
6. $d[w] \leftarrow min\{d[w], d[v] + cost(v, w)\}$;
7. $v_{min} \leftarrow w$ that gives $min\{d[w] \mid w \in V_{i-1}^h\}$;
8. $SV \leftarrow \{v_{min}\}$;
9. **if** $(\mid G_{i-1}^h \mid > c_1)$ **and** $(h \leq c_2)$ **then** $Dynamic(G_{i-1}^h, SV)$;
10. **else** Solve the GSS for G_{i-1}^h;
11. **}**
12. **}**

/***** main program *****/

13. **for** $v \in V$ do $d[v] \leftarrow \infty$;
14. $d[s] \leftarrow 0$;
15. Dynamic(G,$\{s\}$);

Definition 1. *After h^{th} order decomposition, the degree of cyclicity $cyc^h(G)$ is the maximum cardinality of the h^{th} order sc-components of G.*

Lemma 4.2. *After h^{th} order decomposition, let $\rho = cyc^h(G)$, then we can solve the SSSP problem for G in $O(hm + nlog\rho)$ time complexity as each edge is examined at most h times.*

If the $cyc^h(G)$ is a constant value after a constant h order decomposition, we say graph G is nearly acyclic. According this new definition, the original $cyc(G)$ can be described as the maximum cardinality of the 1^{st} order sc-components of G.

4.3 Further Improvement

Let us assume that when SV gets larger, we can break G into more sc-components. As a result, $\mid G_{i-1}^h \mid$ can reach c_1 from above faster. We identify several other minima at line 7 of Algorithm 4.3 while we are searching for the pseudo source. Therefore, in some cases, we can improve the efficiency of the Algorithm 4.3 by enlarging SV (see Algorithm 4.4), and SV can be seen as a set of pseudo sources. In Algorithm 4.4, we initialize an array SV at line 7, and we assume the size of SV is a fixed constant, say L.

Algorithm 4.4. Further improved algorithm of the higher-order approach

1. **procedure** Dynamic(G, SV) **{**
2. Call $SDFS(G, SV)$ to compute h order sc-components $V_r^h, V_{r-1}^h, ..., V_1^h$;
3. **for** $i \leftarrow r$ to 1 **do {**
4. **for** V_j^h such that $(V_i^h, V_j^h) \in \widetilde{E}^h$ **do**
5. **for** $v \in V_i^h$ **do for** $w \in V_j^h$ **do**
6. $d[w] \leftarrow min\{d[w], d[v] + cost(v, w)\}$;
7. Run the GSS on V_{i-1}^h until the size of the solution set S for V_{i-1}^h becomes L. Let $SV = S$.
8. **if** $(\mid G_{i-1}^h \mid > c_1)$ **and** $(h \leq c_2)$ **then** $Dynamic(G_{i-1}^h, SV)$;
9. **else** Solve the GSS for G_{i-1}^h;
10. **}**
11. **}**

/***** main program *****/

12. **for** $v \in V$ **do** $d[v] \leftarrow \infty$;
13. $d[s] \leftarrow 0$;
14. $Dynamic(G, \{s\})$;

Chapter V

Hierarchical Approach

We have mentioned earlier that there are two different measures of what a nearly acyclic graph is: (1) Takaoka gives a definition of acyclicity — the degree of cyclicity of a graph G, $cyc(G)$, is defined by the maximum cardinality of the strongly connected components (sc-components) of G. When the $cyc(G)$ is small, he categorizes the given graph as a nearly acyclic graph [23]. (2) Saunders states that a nearly acyclic graph is a graph that contains relatively few acyclic sub-graphs, each sub-graph of which is dominated by a vertex, called a *trigger* [21]. Obviously, removal of triggers cuts all cycles in the graph. In this chapter, we combine the two measures of near acyclicity, and use a *hierarchical* decomposition to preprocess the graph in $O(m)$ time. Using this preprocessing, a new SSSP algorithm has $O(m + r\log l)$ time complexity, where r is the size of the 1-dominator set, and l is the size of the largest sc-component of 1-dominator triggers. In the first section, we introduce the combined measurement, called a *hierarchical decomposition*. Then, in the second section, we introduce an SSSP algorithm using the result of the *hierarchical decomposition*.

5.1 Hierarchical Decomposition

When we decompose a graph using 1-dominator decomposition (see left picture in Figure 5.1), a degenerated graph (see right picture in Figure 5.1) consists of acyclic structures, denoted by AC. Conceptually we first obtain the 1-dominator set, and degenerate the graph G to G' where the vertices of G' are the triggers in 1-dominator decomposition and an edge from u to v in G' exists if an edge exists from some vertex in $AC[u]$ to v, where $AC[u]$ is an acyclic structure that u belongs to. We sometimes refer to this edge in

G' as a pseudo edge. Then we search for sc-components in the degenerated graph G' (see Figure 5.2). Obviously the maximum size of the sc-components in the above is good enough for the size of the priority queue in the SSSP algorithm.

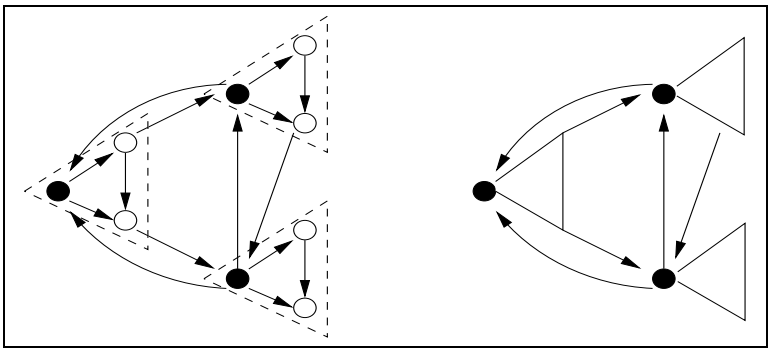

Figure 5.1: A graph with combined nearly acyclic structures

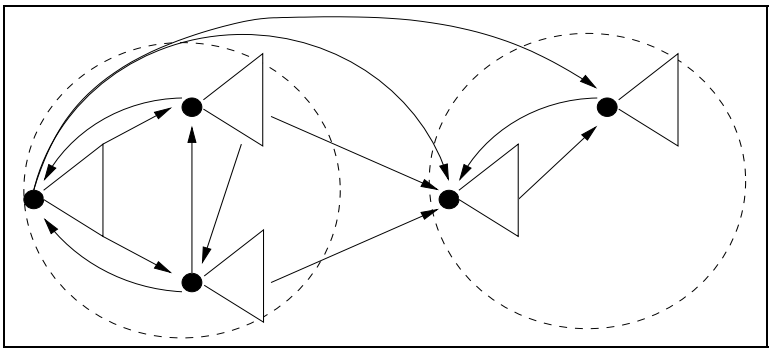

Figure 5.2: A graph with combined nearly acyclic structures, $r = 5$ and $l = 3$.

In the real programming, Algorithm 5.1 modifies Tarjan's algorithm for sc-components (see Algorithm 2.3). We call Algorithm 5.1 *hierarchical depth first search* (HDFS). Algorithm 5.1 calls the algorithm of 1-dominator decomposition (Algorithm 3.2) for triggers as a subroutine. Specifically it calls *AcyclicSet* at line 1.1, and only puts the trigger vertex v into a first-in-last-out stack *STACK* (line 4). If v can reach another trigger vertex w with a lower visit number, we record this reachability in an array *lowLink*[]. When the depth-first-search finishes, the trigger vertices recorded in *STACK* will

Table 5.1: Notations used in algorithms and proofs (sort alphabetically)

A	an acyclic part computed by function *AcyclicSet*()
$AC[v]$	reference value pointing to an acyclic structure dominated by vertex v
B	a boundary vertex set
$BS[v]$	reference value pointing to a boundary vertex set of an acyclic structure $AC[v]$
C	an sc-component computed by function *HierarchySets*()
c	count of the number of visited vertices
$cost(v,w)$	cost of edge $(v \rightarrow w)$
$d[v]$	distance of a path to vertex v
$IN(v)$	$\{w \mid (w \rightarrow v) \in E\}$
$inCount[v]$	number of untraversed incoming edges of vertex v
L	a vertex set maintains all vertices visited in function *AcyclicSet*
$lowLink[v]$	pointing to the root of an sc-component which v belongs to.
$OUT(v)$	$\{w \mid (v \rightarrow w) \in E\}$
p	count of the number of sc-components
$SC[p]$	reference value pointing to an sc-component sorted at topological order p
$STACK$	first-in-last-out stack
$visitNum[v]$	visited order of vertex v

be popped out until a root vertex u if $visitNum[u] = lowLink[u]$, to form an sc-component (line 11). See Figure 5.2 for an example, where the degree of cyclicity of the graph G' is 3.

In order to implement this approach, the visit number $visitNum$ and the low-link number, $lowLink$, of triggers must be computed. In Algorithm 5.1, global variables c and p refer to how many vertices are visited and how many sc-components are identified respectively. Other variables are explained in Table 5.1.

Generally speaking, the Algorithm 5.1 implements the depth first search algorithm in an acyclically decomposed graph. That is, at line 1.2 it first identifies an acyclic sub-graph, and get a set of boundary vertices. Then, the acyclic sub-graph is treated as an entity.

Algorithm 5.1. Hierarchical Depth First Search

function *HierarchySets*(v_0)**{**

1. Vertex Set $C, AC, SC, STACK$

1.1. **procedure** *hdfs*(v)**{**

1.2. (A, B)←*AcyclicSet*(v);

1.3. Let $AC[v]$ refer to A;

1.4. **for** all $u \in A$ **do** $vertexType[u] \leftarrow$*non-trigger*;

1.5. $vertexType[v] \leftarrow trigger$;

2. $visitNum[v] \leftarrow c$; $c \leftarrow c + 1$;

3. $lowLink[v] \leftarrow c$;

4. $STACK \leftarrow STACK + \{v\}$;

5. **for** all $w \in B$ **do**

6. **if** $visitNum[w] = 0$ **then do {**

7. $hdfs(w)$; // *search from unvisited* $w \in B$

8. $lowLink[v] \leftarrow \min(lowLink[v], lowLink[w])$;

9. **}**

10. **else if** $visitNum[w] < visitNum[v]$ **and** $w \in STACK$ **then**
$lowLink[v] \leftarrow \min(lowLink[v], visitNum[w])$; // *update lowLink[v] from connected triggers*

11. **if** $lowLink[v] = visitNum[v]$ **and** $vertexType[v] = trigger$ **do {**

12. $C \leftarrow$ {pop up vertices from $STACK$ until v};

13. $p \leftarrow p + 1$; // *count sc-components*

14. Let $SC[p]$ refer to C;

15. **}**

16. **}**

/***** main program *****/

17. $AC \leftarrow \emptyset$, $SC \leftarrow \emptyset$, $\leftarrow \emptyset$;

18. **for** all $v \in V$ **do {**

18.1. $inCount[v] \leftarrow |IN(v)|$;

18.2. $vertexType[v] \leftarrow$*unknown*;

18.3. $visitNum[v] \leftarrow 0$; $lowLink[v] \leftarrow \infty$; **}**

18.4. $p \leftarrow 0$; $c \leftarrow 1$;

19. **for** all *unvisited* $v_0 \in V$ **do** *hdfs*(v_0);

19.1. **return** (AC, SC, p);

Let us look at the detail of Algorithm 5.1. As we just described, the depth first search for sc-component is build upon acyclic components (lines 1.2 - 1.5). The algorithm marks the triggers of the acyclic components at lines 2-3, and then add the triggers into a stack, called STACK. The adjacent vertices of an acyclic component are the vertices in the boundary set B. If any one of vertices in B is not visited yet, it continues the depth first search (lines 5-7),

In order to computer sc-components on the top of decomposed acyclic entities, triggers represent the acyclic entities in the Algorithm 5.1. At line 4, a trigger is added into a stack, STACK. At line 8 and line 10, the $lowLink[v]$ values of the triggers are updated. When a root of an sc-component has been identified at line 11, an sc-component is identified on the top of the acyclic sub-graphs. Then we remove the vertices from the STACK until the root of the sc-component (lines 12 and 13), In such an sc-component of triggers, we can have the number of triggers in an sc-component, say l_i. Eventually, let

$$l = Max\{l_1, l_2, ..., l_p\}$$

Then we will have $l = cyc(G')$, where $cyc(G')$ is the degree of cyclicity of the degenerated graph G'. The time for Algorithm 5.1 is $O(m)$, as each edge is examined only once.

5.2 Using Hierarchical Decomposition to Compute Shortest Paths Efficiently

We have introduced Algorithm 3.3 in Chapter III which is an SSSP algorithm using acyclic decomposition, and another SSSP algorithm Algorithm 3.1 using sc-components decomposition. Since we combine the acyclic and sc-component decomposition methods, a new SSSP algorithm (Algorithm 5.2) modifies Algorithm 3.3, using some concepts of Algorithm 3.1.

Algorithm 5.2 uses the topologically sorted sc-components and the topologically sorted vertices in each acyclic structure computed by Algorithm 5.1. Obviously only trigger vertices in an sc-component will be added into the frontier set F in a topological order (lines 11-14). That means the non-

trigger vertices will not be involved in the delete-min operations. The distance values $d[v]$ of the non-trigger vertices in an acyclic sub-graph will be decreased straightaway when their associated trigger vertices reach the minimum values.

Algorithm 5.2. SSSP Algorithm Using Hierarchical Decomposition

1. **procedure** *decreaseKey*(u) **{**
2. **for** each $v \in AC[u]$ in topological order **do**
3. **for** each $w \in OUT(v)$ **and** $w \notin S$ **do**
4. $d[w]$=Min$\{d[w],\ d[v]+cost(v,\ w)\}$;
5. **}**

/***** main program *****/

6. **for** all $v \in V$ **do** $d[v] \leftarrow \infty$;
7. the source vertex s and $d[s] \leftarrow 0$;
8. solution vertex set $S \leftarrow \emptyset$;
9. $(AC, SC, p) \leftarrow HierarchySets(s)$;
10. **if** s is not a trigger **then** *decreaseKey*(s);
11. **for** $i \leftarrow p$ to 1 **do {**
12. front vertex set $F \leftarrow \emptyset$;
13. **for** $v \in SC[i]$ **do**
14. **if** v is a *trigger* **then** insert v into F;
15. **while** $F \neq \emptyset$ **do {**
16. $d[u]$ = Min$\{d[u] \mid$ all $u \in F\}$;
17. $F \leftarrow F - u$; // *delete-min*
18. $S \leftarrow S + \{u\}$;
19. *decreaseKey*(u);
20. **}**
21. **}**

In Algorithm 5.2, we assume that sc-components are topologically sorted in the order $p,\ldots, 1$. That is, if there is an edge$(v \rightarrow w)$, $v \in SC[i]$, $w \in SC[j]$, then $i > j$. Let us assume that an acyclic component A has k vertices, and these vertices are topologically sorted in the order $v_1,\ldots, v_k$. That is, $\forall 1 \leq i$,

$j \leq k$ if$(v_i,\ v_j) \in E \Leftrightarrow i < j$. Here, vertex v_1 is the trigger vertex of A. The computing time is

$$O(m + \Sigma_{i=1}^{p} l_i \log l_i)$$

with condition that $l_i \leq l$ and $\Sigma_{i=1}^{p} l_i = r$. This time is maximized when we set as many l_i's to l as possible, and we have the time bounded by $O(m + r \log l)$ from Lemma 1.

Chapter VI

1-2-Dominator Sets

In a 1-dominator set introduced by Saunders and Takaoka [20], intuitively speaking an acyclic structure is an acyclic sub-graph, which is dominated by a vertex called a *trigger*. Any vertex in the sub-graph can only be reached from outside through the trigger. We say that the trigger dominates this acyclic structure. A 1-dominator set is the set of such acyclic structures.

In this chapter, we generalize the 1-dominator set to a *multi*-dominator set, denoted by k-dominator set. In this chapter, we only consider k-dominator sets such that $k \leq 2$ to demonstrate research on the *multi*-dominator sets. In a 2-dominator set, two triggers cooperatively or independently dominate an acyclic structure, and acyclic structures in the 2-dominator set are larger than or equal to the acyclic structures in the 1-dominator set. As a result, we will require less triggers to decompose the graph than a 1-dominator decomposition does. Considering efficient shortest path algorithms only do delete-min operations on triggers, fewer triggers can reduce the time of computing the shortest paths. The following section reviews two approaches to 2-dominator sets proposed by Saunders [21]. Then, in the second and third sections, we introduce a new method and algorithms for computing *multi*-dominator sets.

6.1 2-Dominator Sets

We first analyze two approaches of computing 2-dominator sets proposed by Saunders [21]. We will soon see that there are two major difficulties: (1) How to find a best match of two 1-dominator triggers which can produce a maximal acyclic structure. (2) How to identify the relationship of 2-dominator triggers and 2-dominator non-triggers.

In order to decompose a graph by the multi-dominator approach, one

greedy algorithm was proposed by Saunders [21]. Let us call the algorithm a *Ranking Solution* where each distinct pair of 1-dominator triggers will be computed, and ranked according to the size of the acyclic structure dominated by each trigger pair. Then, the algorithm extracts acyclic structures from the largest ones until the whole graph is covered.

In the *Ranking Solution*, it first computes the 1-dominator trigger set, say T_1. Then, for all $(u_1, u_2) \in T_1 \times T_1$ it computes a possible 2-dominator acyclic structure $Cover(u_1, u_2)$ in which vertices u_1 and u_2 cooperatively dominate some non-trigger vertices or 1-dominator triggers. All the 2-dominator acyclic sets are sorted in a queue Q with the sizes of the sets as keys. That is, the structure $Cover(u_1, u_2)$ is sorted in $Q[i]$ where $i = |Cover(u_1, u_2)|$ and i is the number of vertices in the 2-dominator acyclic set $Cover(u_1, u_2)$.

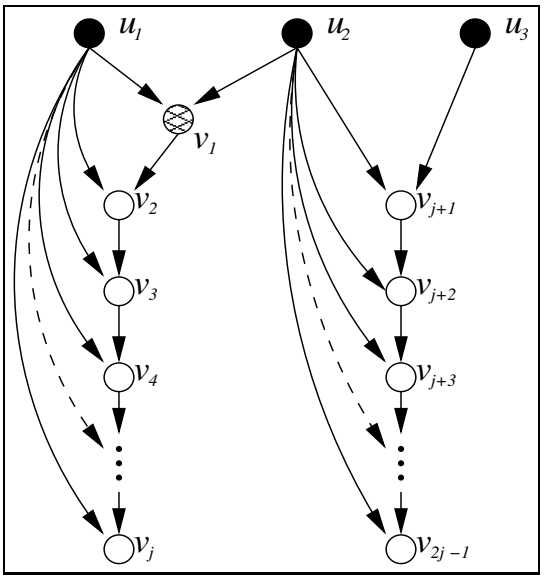

Figure 6.1: Optimal solution is $\{(u_1, v_1), (u_2, u_3)\}$. The dotted-lines indicate the starting endpoints have outgoing edges to every vertex not presented.

After the ranking process, the solution picks up the largest 2-dominator acyclic structure, say $Cover(u_1, u_2)$, listed in Q. The vertices u_1 and u_2 are finalized as 2-dominator triggers, and other vertices in the structure $Cover(u_1, u_2)$ are non-triggers. Then, for every vertex $v \in Cover(u_1, u_2)$, if $Cover(v, w)$ for some $w \in V$ is one of 2-dominator acyclic structures listed in queue Q, the acyclic structure $Cover(v, w)$ will be removed from Q straight-

away. Obviously, the *Ranking Solution* needs to compute $Cover(u_1, u_2)$ for all $(u_1, u_2) \in T_1 \times T_1$. The time complexity becomes $O(mn^2)$.

This solution can solve the first difficulty. However, in the worst case, the number of 2-dominator triggers are n times the optimal solution:

$$Ranking(I) = O(n)Optimal(I).$$

For example, in Figure 6.1 the optimal solution is $\{u_1, v_1, u_2, u_3\}$. However, $Cover(u_1, u_2) = \{u_1, u_2, v_1, ..., v_j\}$ and $\mid Cover(u_1, u_2) \mid = j$. $Cover(u_2, u_3)$ $= \{u_2, u_3, v_{j+1}, ..., v_{2j-1}\}$ and $\mid Cover(u_2, u_3) \mid = j - 1$. According to the *Ranking Solution*, the 2-dominator set is $\{u_1, u_2, u_3, v_{j+1}, ..., v_{2j-1}\}$. Thus, when $j = n$, Ranking(I) = $\frac{n}{4}$Optimal(I) = $O(n)$Optimal(I).

The worst-case time complexity of the algorithm is $O(mn^2)$ [21], and in the worst case the number of 2-dominator vertices computed by the *Ranking Solution* has complexity $O(n)$ times the optimal result, Therefore, the *Ranking Solution* is not a theoretically good solution for computing multi-dominator sets.

The second difficulty was encountered when Saunders was trying to avoid the first difficulty. He proposes another solution, called a *General Solution*, where again for all $(u_1, u_2) \in T_1 \times T_1$ it computes possible 2-dominator acyclic sets $Cover(u_1, u_2)$. Then, for all $(v, w) \in Cover(u_1, u_2) \times Cover(u_1, u_2)$ will be *marked* as 2-dominator trigger pair like (u_1, u_2), or 2-dominator non-trigger pair like (v, u_2) such that $v \in Cover(u_1, u_2)$ and $v \neq u_1$.

But, 2-dominator acyclic sets will not be ranked and extracted according to their associated sizes of acyclic structures. For example, when the pair (u_1, u_2) in Figure 6.1 is computed and *marked* as 2-dominator triggers, then pairs of vertices dominated by u_1 and u_2 are *marked* as *non-trigger pairs*, like (v_1, v_2), (v_1, v_3), (v_1, v_4), ..., (v_2, v_3), ..., (v_{j-1}, v_j). Obviously, it is very costly to repeatedly *mark* all the *non-trigger pairs* for each 2-dominator acyclic structure. In the worst case, the *General Solution* takes $O(mn^4)$ time.

Another problem is that in the *General Solution* some vertices may overlap in several acyclic structures. That is, a vertex v is a dominating trigger in an acyclic structure $A_{(v,w)}$ and and another acyclic structure $A_{(v,l)}$. Or, the vertex v is a trigger in the acyclic structure $A_{(v,w)}$ but is a non-trigger vertex

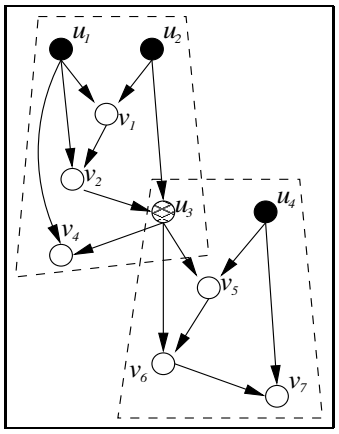

Figure 6.2: Two 2-dominator acyclic structures overlap at vertex u_3.

in another acyclic structure $A_{(l,u)}$ (see Figure 6.2). Thus, the 2-dominator produced by this method is not disjoint.

The time complexity of Algorithm 6.2 is $O(n^4)$ [21], and the result is set-wise unique for the graph. Considering the alternative triggers problem, $General(I) \leq 2 \times Optimal(I)$ [21].

To improve the 2-dominator decomposition, we will look at a new framework. In the new framework, we will have a mixed 1-dominator and 2-dominator vertex set . Let us call it a 1-2-dominator set. The following section gives the detail of this idea.

6.2 Defining 1-2-Dominator Sets

From the review of Saunders' work in the above section, we can see that the above framework of the 2-dominator set is not very efficient for designing the *multi*-dominator set. Therefore, we present a new framework, a 1-2-dominator set. In the 1-2-dominator set, one or two triggers cooperatively dominate an acyclic structure in a graph. Let r' be the number of triggers in the 1-2-dominator set, and let r be the number of triggers in the 1-dominator decomposition. We present algorithms to achieve $O(m+r^2)$ time to compute the 1-2-dominator set. We note that r' is not guaranteed to be smaller than that in a 2-dominator set.

We can run SSSP algorithms using the results of the 1-2-dominator set.

In the worst case, two trigger vertices u and v may cooperatively dominate exactly the same acyclic part of the 1-dominator decomposition. Then, in the worst case, a single source computation using the 1-2-dominator set will scan each acyclic structure up to two times. The corresponding worst-case time complexity of the algorithm is $O(2m + r'\log r') = O(m + r'\log r')$, where constant 2 is emphasized in the left-hand side.

There are two essential concepts in the 1-2-dominator set. Firstly, let us assume that triggers A and B dominate an acyclic structure. Non-trigger vertices in the acyclic structure are marked by the names of the associated triggers A and B. We say that these non-trigger vertices appear in a *territory* of A and B, the *territory* being labeled AB. The *territory* concept saves computing time in identifying the relationships of trigger candidates of a distinct pair. That is, the *General Solution* marks each pair in a *territory* to avoid repeating computations of pairs of non-trigger vertices. The *marking* process is time consuming. Instead of marking each pair of non-trigger vertices, we label each non-trigger vertex by their dominators, e.g., with AB mentioned above. If two vertices have the same label, it means they are in the same *territory*.

In this section, characters h, i, j, k, r will be used to represent numbers. Characters l, t, u, v, w will be used to represent vertices of a given graph $G = (V, E)$.

All the acyclic structures are *complete acyclic structures* (see A_u in the upper picture of Figure 6.3) in the 1-dominator decomposition. That is, an acyclic structure A_u contains all vertices that can be dominated by associated trigger u. A complete acyclic set $A_u \subseteq V$ satisfies the following requirements:

— $A_u - \{u\}$ is acyclic. That is, the graph induced by vertices in $A_u - \{u\}$ contains no cycles.
— $u \in A_u$.
— For any w, if $IN(w) \subseteq A_u$, then $w \in A_u$.

To make the definition simple, we introduce a degenerated graph $G' = (V', E')$ derived from triggers (see the lower picture in Figure 6.3), $V' = \{A_u, A_v, ..., A_w\}$ where $\{A_u, A_v, ..., A_w\}$ are vertices degenerated from the 1-

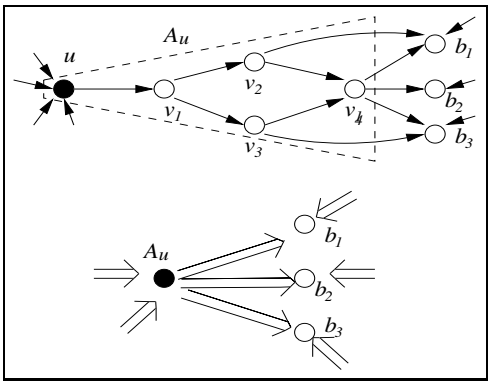

Figure 6.3: Degenerating an acyclic structure. Arrows "⇒" represent some edges to or from the degenerated vertex A_u.

dominator acyclic structures in the graph. E' is the set of edges between $A_u, A_v, ..., A_w$, sometimes we call those edges the "*pseudo edges*".

The collection of 1-dominator acyclic sets is set-wise unique, and that has been proved by Saunders and Takaoka [22]. So, the degenerated graph $G' = (V', E')$ is unique for the given graph G. Then, the following argument will be based on G', and prove that the collection of 1-2-dominator acyclic sets is also unique. Since the proof is based on G', in the following argument a vertex indicates a degenerated vertex in V', and an edge indicates a pseudo edge in E' (see the lower picture in Figure 6.3).

Based on the degenerated graph $G' = (V', E')$, an acyclic structure $A_{(v,w)}$ maintains vertices dominated cooperatively by triggers A_v and A_w. Let us call $A_{(v,w)}$ the 2-dominator acyclic structure. A combined acyclic set $\widetilde{A}_{(v,w)}$ is a set of triggers A_v and A_w and acyclic structures dominated by A_v or A_w or by both. Let us call $\widetilde{A}_{(v,w)}$ the acyclic set. $A_{(v,w)}$ and $\widetilde{A}_{(v,w)}$ are defined algorithmically as:

Definition 1.

— Initially, $A_{(v,w)} = \emptyset$ and $\widetilde{A}_{(v,w)} = \{A_v, A_w\}$;

— Iteratively $\widetilde{A}_{(v,w)} \leftarrow A_{(v,w)}$

$$A_{(v,w)} \leftarrow \widetilde{A}_{(v,w)} + \{A_t \mid A_t \in V', IN(A_t) \subseteq \widetilde{A}_{(v,w)}\}$$

until $\widetilde{A}_{(v,w)} = A_{(v,w)}$.

A 1-dominator set is set-wise unique, and that has been proved by Saunders and Takaoka [22]. As a 2-dominator acyclic structure increases in size,

the number of triggers decreases. In the following, we shall prove that $\widetilde{A}_{(v,w)}$ is a complete acyclic set similar to that we have defined in the 1-dominator decomposition on the previous page.

By indexing the iteration in the **Definition 1**, the definition for $\widetilde{A}_{(v,w)}$ is written as $\widetilde{A}_{(v,w)} = \widetilde{A}_{(v,w)}^{(0,\ldots,\beta(v,w)-1)}$ where the value $\beta(v,w)$ is such that $\widetilde{A}_{(v,w)}^{\beta(v,w)}$ is the boundary set of $\widetilde{A}_{(v,w)}$, and $\widetilde{A}_{(v,w)}^{(j,\ldots,k)} = \bigcup_{i=j}^{k} \widetilde{A}_{(v,w)}^{(i)}$ with $\widetilde{A}_{(v,w)}^{(i)}$ defined as follows:

— $\widetilde{A}_{(v,w)}^{(0)} = \{A_v, A_w\}$
— $\widetilde{A}_{(v,w)}^{(i+1)} = \{t \mid IN(t) \cap \widetilde{A}_{(v,w)}^{(i)} \neq \emptyset \text{ and } IN(t) \subseteq \widetilde{A}_{(v,w)}^{(0,\ldots,i)}\}$.

This definition is depicted in Figure 6.4.

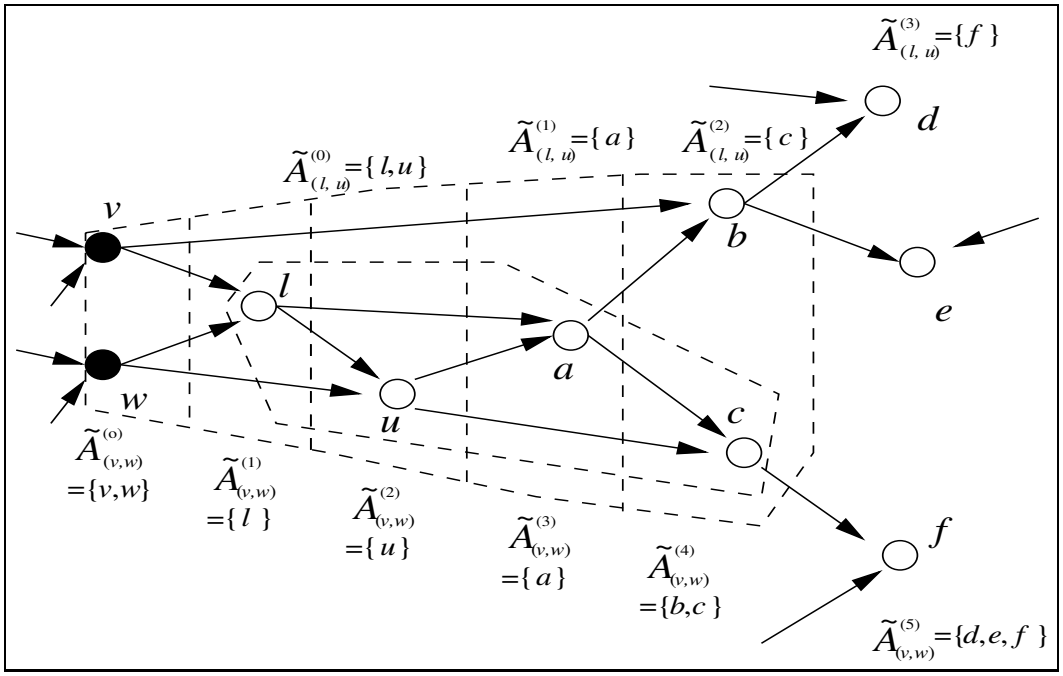

Figure 6.4: $\widetilde{A}_{(v,w)}$ is maximal. But $\widetilde{A}_{(l,u)}$ is not maximal and $\widetilde{A}_{(l,u)} \subseteq \widetilde{A}_{(v,w)}$.

Lemma 6.1. *If $A_l \in \widetilde{A}_{(v,w)}^{(j)}$ and $A_u \in \widetilde{A}_{(v,w)}^{(k)}$ for some j and k such that $j \leq k$ then $\widetilde{A}_{(l,u)}^{(i)} \subseteq \widetilde{A}_{(v,w)}^{(j+i)} \cup \widetilde{A}_{(v,w)}^{(k+i)}$ for all $0 \leq i < \beta(v,w)$.*

Proof(By induction) *Basis* $i = 0$ $\widetilde{A}^{(0)}_{(l,u)} = \{A_l, A_u\}$. It is given that $\{A_l\} \subseteq \widetilde{A}^{(j)}_{(v,w)}$ and $\{A_u\} \subseteq \widetilde{A}^{(k)}_{(v,w)}$. Thus, $\widetilde{A}^{(0)}_{(l,u)} \subseteq \widetilde{A}^{(j)}_{(v,w)} \cup \widetilde{A}^{(k)}_{(v,w)}$

Induction: It is given that $j \leq k$. Above induction provides the assumption that $\widetilde{A}^{(h)}_{(l,u)} \subseteq \widetilde{A}^{(j+h)}_{(v,w)} \cup \widetilde{A}^{(k+h)}_{(v,w)}$ for $0 \leq h \leq i$, from which it follows that $\widetilde{A}^{(0,\ldots,i)}_{(l,u)} \subseteq \widetilde{A}^{(j,\ldots,k+i)}_{(v,w)}$. Thus $\widetilde{A}^{(0,\ldots,i)}_{(l,u)} \subseteq \widetilde{A}^{(0,\ldots,k+i)}_{(v,w)}$. Obviously, there should be no other vertex $A_t \in \widetilde{A}^{(h)}_{(l,u)}$ such that $j \leq h \leq k$ and $A_t \notin \{A_l, A_u\}$ because A_t in this position can not have incoming edges from A_u. As a result, if A_t only has incoming edges from A_l, then A_t is dominated by A_l. Or A_t has incoming edges from other vertices in $\widetilde{A}_{(v,w)}$ and then $A_t \notin \widetilde{A}_{(l,u)}$.

Now consider the definitions of $\widetilde{A}^{(i+1)}_{(l,u)}$ and $\widetilde{A}^{(k+i+1)}_{(v,w)}$:

— $\widetilde{A}^{(i+1)}_{(l,u)} = \{A_t \mid IN(t) \cap \widetilde{A}^{(i)}_{(l,u)} \neq \emptyset \text{ and } IN(A_t) \subseteq \widetilde{A}^{(0,\ldots,i)}_{(l,u)}\}$
— $\widetilde{A}^{(k+i+1)}_{(v,w)} = \{A_t \mid IN(A_t) \cap \widetilde{A}^{(k+i)}_{(v,w)} \neq \emptyset \text{ and } IN(A_t) \subseteq \widetilde{A}^{(0,\ldots,k+i)}_{(v,w)}\}$.

Given that $\widetilde{A}^{(i)}_{(l,u)} \subseteq \widetilde{A}^{(k+i)}_{(v,w)}$: If $IN(A_t) \cap \widetilde{A}^{(i)}_{(l,u)} \neq \emptyset$, then $IN(A_t) \cap \widetilde{A}^{(k+i)}_{(v,w)} \neq \emptyset$. Similarly, given $\widetilde{A}^{(0,\ldots,i)}_{(l,u)} \subseteq \widetilde{A}^{(0,\ldots,k+i)}_{(v,w)}$: If $IN(A_t) \subseteq \widetilde{A}^{(0,\ldots,i)}_{(l,u)}$, then $IN(A_t) \subseteq \widetilde{A}^{(0,\ldots,k+i)}_{(v,w)}$. Thus, as defined, the set $\widetilde{A}^{(k+i+1)}_{(v,w)}$ contains all vertices in the set $\widetilde{A}^{(i+1)}_{(l,u)}$. That is, $\widetilde{A}^{(i+1)}_{(l,u)} \subseteq \widetilde{A}^{(k+i+1)}_{(v,w)}$. Hence by induction on i: $\widetilde{A}^{(i)}_{(l,u)} \subseteq \widetilde{A}^{(j+i)}_{(v,w)} \cup \widetilde{A}^{(k+i)}_{(v,w)}$ for all $0 \leq i < \beta(v,w)$ and $j \leq k$. $\triangle$

Lemma 6.2. *for all* $(A_l, A_u) \in \widetilde{A}_{v,w} \times \widetilde{A}_{v,w}$ *we have* $\widetilde{A}_{l,u} \subseteq \widetilde{A}_{v,w}$.

Proof A consequence of Lemma 6.1 is that if $\{A_l, A_u\} \subseteq \widetilde{A}_{(v,w)}$, then $\widetilde{A}_{(l,u)} \subseteq \widetilde{A}_{(v,w)}$. Then, for all the possible pairs such that $(A_l, A_u) \in \widetilde{A}_{v,w} \times \widetilde{A}_{v,w}$ we have $\widetilde{A}_{l,u} \subseteq \widetilde{A}_{v,w}$. $\triangle$

Lemma 6.3. *When the algorithm in Definition 1 is finished for all v and w in a given graph and* $\widetilde{A}_{(l,u)}$ *is removed if* $A_l \in \widetilde{A}_{(v,w)}$ *and* $A_u \in \widetilde{A}_{(v,w)}$ *, all the remaining acyclic structures are maximal.*

Proof After the graph decomposition, the 1-dominator acyclic structures are are also maximal, which was proved by Saunders (2004). A consequence of the Lemma 6.2 is that there will be 2-dominator acyclic structures that are maximal, and they satisfy the property: $\widetilde{A}_{(l,u)} \subseteq \widetilde{A}_{(v,w)}$ for all $(A_l, A_u) \in \widetilde{A}_{(v,w)} \times \widetilde{A}_{(v,w)}$ such that $\widetilde{A}_{(v,w)} \neq \widetilde{A}_{(l,u)}$. $\triangle$

The vertices v and w denoting a maximal acyclic structures $\widetilde{A}_{(v,w)}$ are called the 1-2-dominator triggers of $\widetilde{A}_{(v,w)}$.

Lemma 6.4. *The collection of acyclic structures constituting the 1-2-dominator set is unique for a given graph.*

Proof We prove that the 1-2-dominator set is set-wise unique by contradiction. Assume to the contrary that a 1-2-dominator set is not unique. Then, there must be v_0 that satisfies $v_0 \in \widetilde{A}_{(v,w)}$ and $v_0 \in \widetilde{A}_{(l,u)}$ for maximal $\widetilde{A}_{(v,w)}$ and $\widetilde{A}_{(l,u)}$ such that $\widetilde{A}_{(v,w)} \neq \widetilde{A}_{(l,u)}$. Then, either $A_l \in \widetilde{A}_{(v,w)}$ and $A_u \in \widetilde{A}_{(v,w)}$, or $A_v \in \widetilde{A}_{(l,u)}$ and $A_w \in \widetilde{A}_{(l,u)}$. In either case one becomes non-maximal from Lemma 6.2, which is a contradiction. $\triangle$

6.3 1-2-Dominator Set Algorithms

Now, we shall show algorithms for computing a 1-2-dominator set. First we compute a 1-dominator set of a given graph. In order to save the time for computing a 1-2-dominator set, we degenerate each acyclic structure into its associated trigger vertex during the 1-dominator decomposition. In Figure 6.3 the upper picture is degenerated into the lower picture with pseudo edges represented by "⇒". We assign a boundary vertex set $BS[v]$ for the trigger vertices. For a boundary vertex w in a set $BS[v]$, a variable $rem(v,\ w)$ maintains the number of outgoing edges from $AC[v]$ to w, together with w in BS.

Algorithm 6.1 implements this process. It modifies the function *AcyclicSet* of Algorithm 3.2 with line numbers extended with dots (lines 15-15.4). The main program of Algorithm 3.2 is the same. In this algorithm, a vertex set A reserves all the non-trigger vertices dominated by the associated trigger vertex, a vertex set L reserves all vertices visited by the current search, a vertex set B contains all the boundary vertices of an acyclic structure.

Algorithm 6.1. Modified Function for Computing AC and BS

1. **function** *AcyclicSet*(v)**{**
2. VertexSet A, L, B;
   // B is enhanced with rem[]
3. **procedure** *rdfs*(u) **{**
4. $A \leftarrow A + u$;
5. **for** all $w \in OUT(u)$ **do {**
6. **if** $w \notin L$ **then** $L \leftarrow L \cup \{w\}$;
7. $inCount[w] \leftarrow inCount[w] - 1$;
8. **if** $inCount[w] = 0$ **then** *rdfs*(w);
9. **}**
10. **}**
11. $A \leftarrow \emptyset$; $L \leftarrow \{v\}$;
12. $inCount[v] \leftarrow inCount[v] + 1$;
13. *rdfs*(v);
14. VertexSet $B \leftarrow L - A$; // *boundary vertices*
15. **for** each $w \in B$ **do {**
15.1 $rem(v, w) = |IN(w)| - inCount[w]$;//remaining number of untraversed edges
15.2 $BS[v] \leftarrow (w, rem(v, w))$;
15.3 $inCount[w] \leftarrow |IN(w)|$;
15.4 **}**
16. **return** (A, B);
17. **}**

/***** main program *****/

18. **for** all $v \in V$ **do** $vertexType[v] \leftarrow unknown$;
19. **for** all $v \in V$ **do** $inCount[v] \leftarrow |IN(v)|$;
20. $Q \leftarrow \{s\}$;
21. **while** $Q \neq \emptyset$ **do {**
22. Remove the next vertex u from Q;
23. **if** $vertexType[u] = unknown$ **then {**
24. $(A, B) \leftarrow AcyclicSet(u)$;
25. Let $AC[u]$ refer to A;
26. $vertexType[u] \leftarrow trigger$;

```
27.     for all v ∈ A do
            vertexType[v] ←non-trigger;
28.     for all v ∈ B do
29.       if vertexType[v] = unknown and v ∉ Q
            then Add v to Q;
30.   }
31. }
```

After we finish the computation of 1-dominator decomposition, we look up the tables of boundary sets $BS[v]$, for a 1-dominator trigger v, to do the 2-dominator decomposition, rather than traverse the whole graph again.

Algorithm 6.2 implements this design. In Algorithm 6.2, a vertex set T_1 reserves all the 1-dominator trigger vertices. Note that $BS[v]$ in function $gdfs()$ plays the role of $OUT(v)$ in Algorithm 3.2.

From one 1-dominator trigger, say u_1, we look for a partner, say u_2, which can co-dominate some part of the graph. The co-dominated part is given to both $AC[u_1]$ and $AC[u_2]$. Unlocked vertices are labeled by its dominators (line 5) to identify which *territory* they belong to. The generalized depth-first search $gdfs()$ goes over the degenerated graph. If a vertex v has boolean value $border[v] = true$, v has contact with boundary vertices of an acyclic structure, and it can still be a trigger candidate (line 17). Otherwise, if v has boolean value $border[v] = false$, v is in an acyclic structure and has no edge connections with any vertices out side the acyclic structure (line 6).

At line 14, it takes $O(m)$ to complete a 1-dominator set computation. From line 15 to 25, it takes $O(r^2)$ to check all the possible pairs of 1-dominator triggers. The total time spent by $gdfs$ is $O(m')$ where m' is the number of pseudo edges bounded by m. That is because we start a new search from a border vertex and thus pseudo edges are examined at most two times. So, the time complexity of Algorithm 6.2 is $O(m + r^2)$. Algorithm 6.1 decomposes set V into 1-dominator structures. Algorithm 6.2 in fact decomposes V into a mixture of 1-dominator structures and 2-dominator structures. This decomposition is set wise unique as the 1-dominator decomposition.

Algorithm 6.2. 1-2-dominator Set Algorithm

1. **procedure** *gdfs*(u_1, u_2, v) **{** //u_1 and u_2 are possible triggers.
2. **for** all $w \in BS[v]$ **{**
3. $inCount[w] \leftarrow inCount[w] - rem(v, w)$;
4. **if** $inCount[w] = 0$ **do {** // *w is unlocked*
5. w is labeled by its dominators u_1 and u_2.
6. $border[w] = false$;
7. $AC[u_1] \leftarrow AC[u_1] + \{w\}$;
8. $AC[u_2] \leftarrow AC[u_2] + \{w\}$;
9. *gdfs*(u_1, u_2, w);
10. **}**
11. **else** $border[v] = true$;
12. **}**
13. **}**

/***** main program *****/

14. Compute 1-dominator set T_1 by Algorithm 6.1;
15. **for** all $v \in T_1$ **do** $border[v] = true$;
16. **for** each $u_1 \in T_1$ **do{**
17. $inCount[u_1] \leftarrow inCount[u_1] + 1$;
18. **for** all $v \in BS[u_1]$ **do**
19. $inCount[v] \leftarrow inCount[v] - rem(u_1, v)$;
20. **for** $u_2 \in T_1 - \{u_1\}$ **and** u_1, u_2 are not in the same $Territory$ **and** $border[u_2] = true$ **do {**
21. $inCount[u_2] \leftarrow inCount[u_2] + 1$;
22. *gdfs*(u_1, u_2, u_2);
23. **}**
24. **for** all $v \in BS[u_1]$ **do** $inCount[v] \leftarrow |IN(v)|$;
25. **}**

To solve the SSSP problem, we can use Algorithm 3.3 with the set of remaining triggers computed by Algorithm 6.2. Algorithm 3.3 will perform $decreaseKey$ operations on 1-dominator structures once each, and twice on 2-dominator structures.

Once the 2-dominator set has been computed, we can repeatedly use it for computing SSSPs. When r' is much smaller than r, the decomposition time of the 2-dominator set will be balanced off by the time saved from repeatedly computing SSSPs, when only edge costs change in the same graph structure.

Chapter VII

Evaluation

The new shortest path algorithms presented in this thesis are theoretically more efficient than other algorithms described in the review of the related work in Chapter III for solving the SSSP problem on suitable nearly acyclic graphs. New algorithms offer potential improvements on the running time in practice. Therefore, the aim of the experiments presented in this chapter is to see what kind of practical performance improvement is achieved on certain kind of graphs. In this chapter, we will look at what exact improvements are possible using comparisons of computing time of different algorithms. In the first section of this chapter, we explain the methodology of the experiments. Then, in the second section of this chapter, we present the detail of the comparisons, and analyze the experimental results.

7.1 Experimental Setup

The algorithms introduced in this thesis have been implemented and run on a computer. The computing time measured provides comparisons of how well they perform in practice. Some factors that can affect the performance of the algorithms need to be addressed first. The way of generating input data also need to be discussed before we can analyze the experiment results.

7.1.1 Factors Affecting the Performance of Algorithms

Many factors associate with SSSP algorithms, and they can affect the performances of the algorithms. First of all, algorithms are designed for certain types of graphs — directed or undirected, positive-valued or negative-valued or integer edge costs. Secondly, algorithms may work for a certain family of graphs, such as acyclic graphs, planar graphs, strongly connected graphs or

nearly acyclic graphs. Thirdly, algorithms are specified for different shortest path problems — single-source, single-pair or all-pairs. Finally, algorithms may work under different computational models — the comparison-addition model or word RAM (Random Access Machine) model, and whether the result is associated with the worst-case or average-case time complexity [21, 4, 7]. We have given details of these factors in the Chapter II.

Algorithms presented in this thesis are designed for directed nearly acyclic graphs with non-negative real-valued edge costs. The algorithms solve the single-source shortest path problem under the comparison-addition model using the worst-case time complexity analysis.

In order to develop faster algorithms for solving the SSSP problem, new parameters are introduced into the worst-case time complexity, which relates to some measurable property in a graph, for example, parameter k in Takaoka's sc-component algorithm with time complexity $O(m + n\log k)$ where k is the largest size of sc-component in the graph. Algorithms developed in this thesis use this approach, too. In the *higher-order* algorithm with $O(hm+n\log\rho)$ time complexity, new parameter ρ is the number of vertices of the largest sc-component of a degenerated graph which has been decomposed h times. In the *hierarchical* algorithm with $O(m+r\log l)$ time complexity, new parameter l is the size of the largest sc-component of a degenerated graph derived from 1-dominator triggers and edges between 1-dominator acyclic structures. In the *1-2-dominator* algorithm with $O(m + r'\log r')$ time complexity, parameter r' is the number of triggers in 1-2-dominator set [21].

The main attribution of new parameters is a reduction in the amount of computation time that is associated with priority queue manipulations [21]. However, we have to preprocess the graph to measure the particular properties of the graph, and then we can determine the values of the new parameters. The preprocessing time may be measured as part of the total time of computing shortest paths. But, the advantage of preprocessing the graph is to reduce the time spent in manipulating the priority queue at the expense of time for preprocessing the graph [21].

If the decomposition time is linear such as 1-dominator decomposition in $O(m)$ time, sc-component decomposition in $O(m)$ time, hierarchical decomposition in $O(m)$ time, and higher-order decomposition in $O(hm)$ time,

then new introduced parameters related to the measured property will be particularly beneficial for SSSP algorithms. For expensive decompositions like 1-2-dominator set in $O(m + r^2)$ time, we may still gain efficiency by repeatedly reusing the decomposition results in solving the SSSP problem.

7.1.2 Generating Graphs

Experimented graphs are line-spanning random graphs. That is, for a graph $G = (V, E)$ with n vertices in V and initially no edges in E, we first add all the edges (v_i, v_{i+1}) for $1 \leq i < n$ into E. Then we repeatedly generate edges $(v \rightarrow w)$ at random such that $v \in V$, $w \in V$, $(v \rightarrow w) \notin E$ and $v \neq w$. Here, we use an edge factor f to specify those randomly added edges. Therefore, the total number of edges $m = (1 + f)n$ [21]. Each edge has a cost between 1 and 100.

In order to compare the various algorithms, we generate some suitable graphs that are composed of a particular graph structure recognized by a specialized algorithm. This will easily demonstrate the improved practical performance provided by a more efficient algorithm though it is rather artificial. This will be particularly true for algorithms based on *higher-order decomposition*. One group of experimental graphs has nearly acyclic structures where the degree of cyclicity is $cyc(G)$ small (see Figure 7.1 for an example). Another group of experimental graphs (see Figure 7.2) is developed upon the graph in Figure 7.1. The third group of graphs has few simple cycles for the graph size (see Figure 7.3 for an example). The fourth group of graphs has combined structures (see Figure 7.4 for an example). In those experiments, graphs have n vertices and $2.8n$ edges. The number of vertices in the graphs start at 2,000 for Figures 7.1 and 7.3, and 2,197 for Figure 7.4 and doubling for successive values of n up until 128,000 for Figures 7.1 and 7.3 and 79,507 for Figure 7.4. The number of vertices in Figure 7.2 starts from 35,000 and then increasing by 15,000 each time up until 170,000. This provides a large enough window to demonstrate the overall trends in algorithm performance.

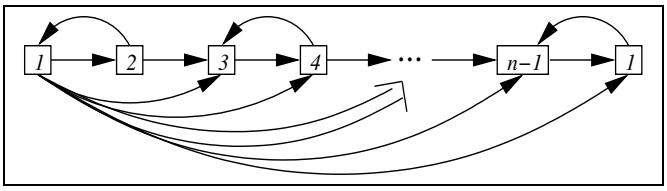

Figure 7.1: Arrow "⇒" indicates *Vertex* 1 has edges to each other node i, 4 $< i < n - 1$, in the graph. Let *Vertex* 1 be the source, $cyc(G) = 2$, $r = n/2$, $l = 2$.

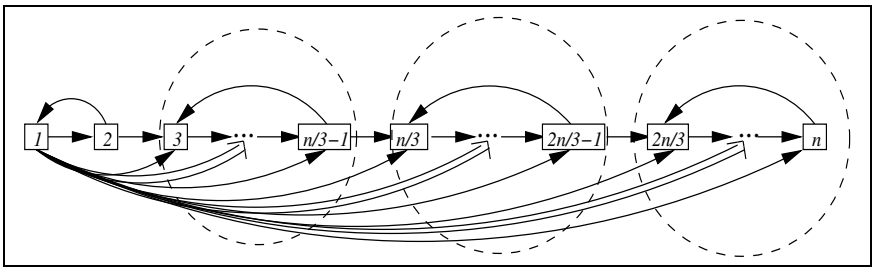

Figure 7.2: Arrow "⇒" indicates *Vertex* 1 has edges to every node unstated. Let Vertex 1 be the source, $cyc^1(G) = n/3$, $cyc^2(G) = 2$.

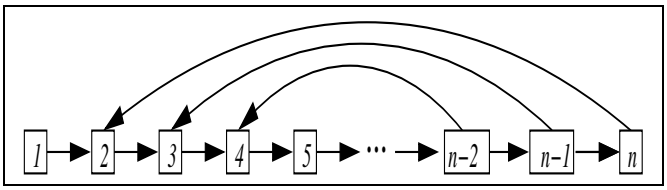

Figure 7.3: Let *Vertex* 1 be the source, $cyc(G) = n - 1$, $r = 3$, $l = 3$.

7.2 Experimental Results and Analysis

The experiments were all performed using a 2.4GHz Intel Pentium 4 computer with 512 MB of RAM, running the Fedora Linux operating system. All algorithm implementations were written in **C** programming language using the same programming style. Implementations were compiled using GNU project compiler **gcc**.

The running time of each algorithm was measured by the amount of CPU time they consumed. But, for sample graphs with the same type and parameters m and n, the final time measurement of algorithm running time was the average running time over 50 sample graphs because of the variation

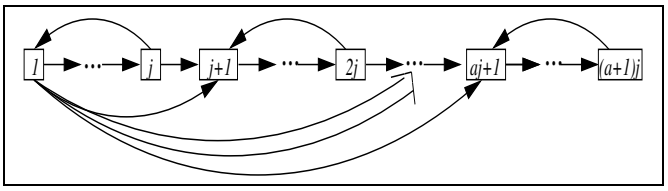

Figure 7.4: $j = \sqrt{n}$ and $a = \sqrt{n}-1$. Arrow "⇒" indicates *Vertex* 1 has edges to every node $ij + 1$, $1 < i < a$, in the graph. Let *Vertex* 1 be the source, $cyc(G) = \sqrt{n}$, $r = \sqrt{n}$, $l = 1$.

Table 7.1: The different algorithms implemented in experiments.

Name	*Description*
Acyclic	Acyclic approach, Saunders & Takaoka (2005)
Dijkstra	Dijkstra's SSSP algorithm
Hierarchical	The new hierarchical approach
Second Order	Higher-Order approach using the second order
SC	Strongly connected approach, Takaoka (1998)

that occurred among randomly generated graphs. In such a way, we can achieve an acceptable accuracy of time measurement within the operating system clock granularity [21].

Five algorithms have been implemented (see Table 7.1) for solving the SSSP problem, and the SSSP computation time has been measured for analysis. Figure 7.5 shows that the new hierarchical approach performs as efficiently as that of the *SC* approach in graphs of Figure 7.1. We call such graphs *SC* approach favored graphs. Figure 7.6 shows that second order decomposition can provide some efficiency improvement in solving SSSP problem for some graphs like the graph in Figure 7.2. Figure 7.7 shows that the new approach performs the SSSP computation as efficiently as that of the acyclic approach in the acyclic approach favored graphs of Figure 7.3. Figure 7.8 shows that the new approach can outperform the other two approaches in graphs with combined nearly acyclic structures (see the graph in Figure 7.4).

For 1-2-dominator algorithms presented in Chapter VI, in Figure 7.9 there are proportions of triggers in 1-dominator sets and 1-2-dominator sets. In the experiments, there are 1,000 vertices in each graph, but the number of edges

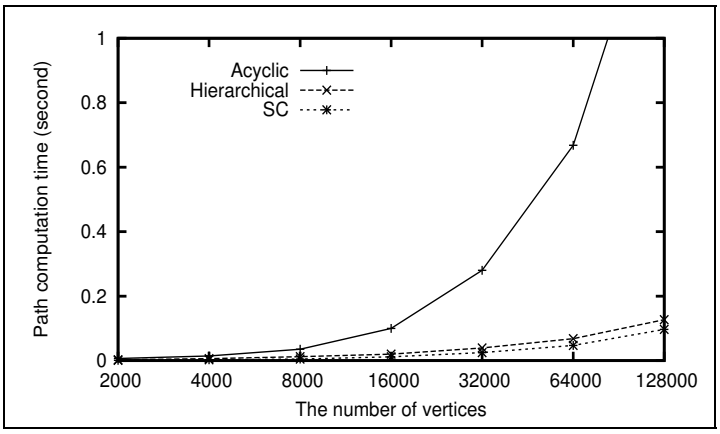

Figure 7.5: Evaluation of algorithms solving SSSP problem for graphs presented in Figure 7.1

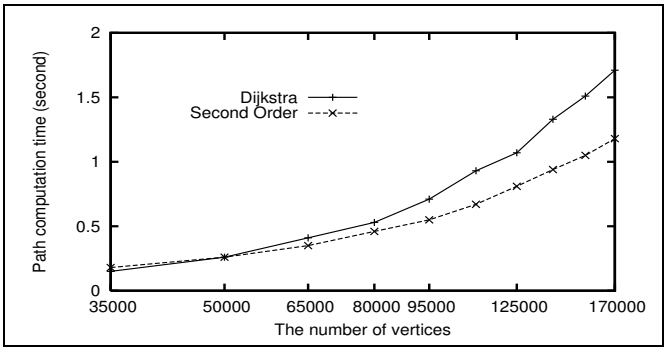

Figure 7.6: Evaluation of algorithms solving SSSP problem for graphs presented in Figure 7.2

varies from 1.1×1,000 to 13.8×1,000. The abscissa shows the edge factor of each graph. From Figure 7.9, we can see that the number of triggers in 1-2-dominator sets is about 20 percent smaller than that in the 1-dominator sets in sparse graphs with edge number between *1.2n* and *4.2n*

In Figure 7.10, there is the time of solving the SSSP problem between using 1-dominator sets and using 1-2-dominator sets. Graphs have 1,000 vertices and edge factors vary from 0.1 to 0.25. The ordinate shows the computation time measured in seconds. We can see from this figure that when the graph is sparse and the edge factor is between 0.005 and 0.25, it reflects the efficiency gained from the reduced number of triggers in 1-2-dominator sets. When the graph becomes denser and the edge factor gets

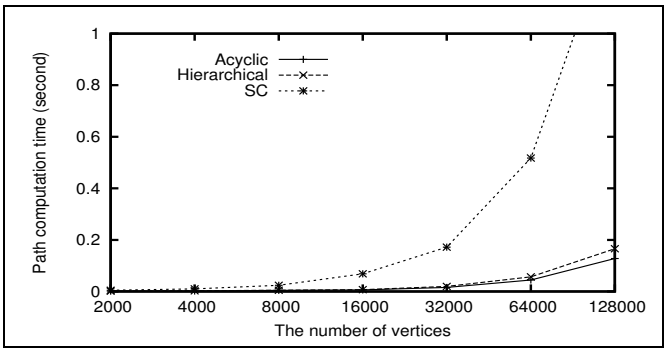

Figure 7.7: Evaluation of algorithms solving SSSP problem for graphs presented in Figure 7.3

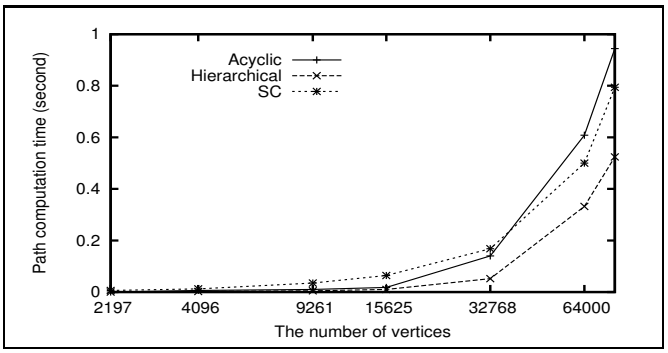

Figure 7.8: Evaluation of algorithms solving SSSP problem for graphs presented in Figure 7.4

closer to 0.25, the difference of computation time gets smaller and smaller until there is almost no difference at 0.25.

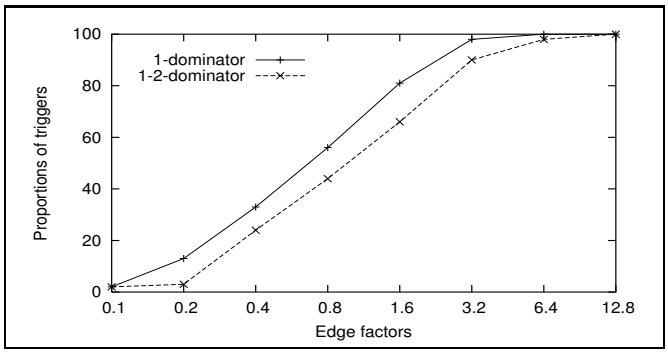

Figure 7.9: The proportion of triggers in 1-dominator sets and 1-2-dominator sets

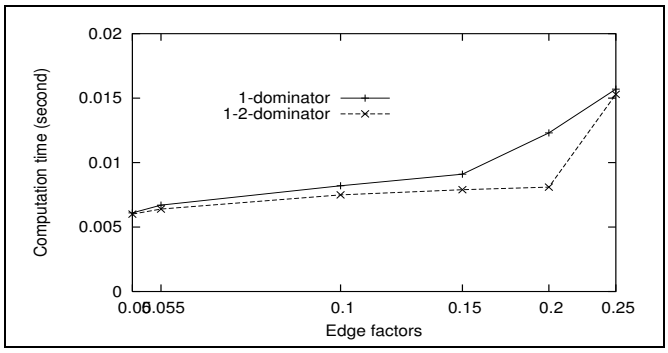

Figure 7.10: the computing time for SSSP problem between using 1-dominator sets and using 1-2-dominator sets in different density graphs

Chapter VIII

Summary and Conclusions

The research presented in this thesis has explored how to solve the single source shortest path problem efficiently for directed nearly acyclic graphs. New algorithms have been developed, and they provide considerable contributions to the existing works that have been done in this area. Section 8.1 is a summary of the different kinds of measures for acyclicity, which have been introduced in articles and this thesis. Then, there is a summary of the new algorithms that have resulted from this research. Section 8.2 points out some possibilities for future research.

8.1 Summary

In 1993 Abuaiadh and Kingston [1] suggested that the inherent complexity of the shortest path problem depends on the cycle structures of a graph as well as on its size. They gave an algorithm with $O(m+n\log t)$ time complexity, where t was the number of delete_min operations needed in the priority queue manipulations. For nearly acyclic graphs, t was expected to be small, so their algorithm could efficiently solve the SSSP problem [1]. However, they did not give a clear definition of t, and the new parameter t had no direct relation with the graph structures. Later in 1994, they introduced another algorithm with time complexity $O(m+k\log k)$, where k was the number of cycles in a graph [3]. This had been improved by Saunders and Takaoka in 2005 (see a research map in Figure 8.1).

Takaoka gave a definition of acyclicity. The degree of cyclicity of a graph G, $cyc(G)$, was defined as the maximum cardinality of strongly connected components (sc-components) of G. When $cyc(G)$ was small, he clarified the given digraph G to be nearly acyclic. When $cyc(G) = k$, he gave an algorithm

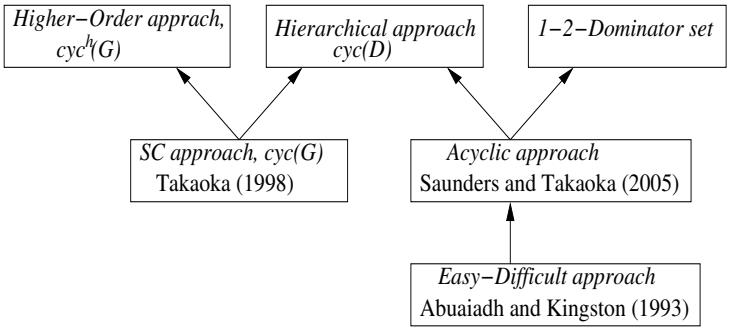

Figure 8.1: A summary of research solving the SSSP problem for nearly acyclic graphs

with O(m+$n$log$k$) time complexity [23].

Saunders and Takaoka offered an acyclic decomposition approach. In this approach, a graph was decomposed into acyclic structures in O(m) time. Each structure was dominated by a trigger vertex. Triggers formed a 1-dominator set. If a nearly acyclic graph had r trigger vertices, they introduced an algorithm with O(m+$r$log$r$) time complexity [22] for solving SSSP problems. The new parameter r represented the number of acyclic structures in a graph.

In this thesis, we first present a *higher-order* approach to decompose a graph into strongly connected components (sc-components) as small as possible. Using higher-order decomposition, we give another definition of acyclicity, which is generalized from the Takaoka's definition [23]. That is, the degree of cyclicity is the maximal cardinality of h^{th} order strongly connected component of a graph G, and the degree of cyclicity is denoted by $cyc^h(G)$. Here, parameter h is the number of times that the graph has been decomposed. When the graph is decomposed into sc-components, for computing the shortest paths we run Dijkstra's algorithm only for each sc-component but not the whole graph. When all sc-components are small and the degree of acyclicity $cyc^h(G) = \rho$, we can efficiently compute SSSPs in $O(hm+n\log\rho)$ time. If we repeatedly use the decomposed graph for solving the SSSP problem, we can balance the time used for preprocessing the graph.

Then, we show that we can compute acyclic components and sc-components of the graph at the same time. The benefit of merging these two graph pre-

processing approaches is to make a superior measure over the two existing measures of nearly acyclic graphs. Let r be the number of trigger vertices in the 1-dominator set, and $l = cyc(G')$ where $cyc(G')$ is the degree of the acyclicity of the degenerated graph G'. In the degenerated graph G', the vertices of G' are the 1-dominator triggers and an edge from u to v in G' exists if an edge exists from some vertex in $AC[u]$ to v, where $AC[u]$ is an acyclic structure that u belongs to. When l or r is small, we say that the graph is nearly acyclic, and we can efficiently solve the SSSP problem for the graph in time $O(m+r\log l)$, which takes advantage of both measures.

We also present a demonstration of a 1-2-dominator set, which can be generalized to a *multi*-dominator set, denoted by k-dominator set. In a 1-2-dominator set, generally speaking, one or two trigger vertices cooperatively dominate an acyclic structure in a graph. Once the 1-2-dominator set has been computed, we can repeatedly use it for computing the shortest paths. When the 1-2-dominator set is much smaller than the size of the 1-dominator set, the decomposition time of the 1-2-dominator set will be balanced off by the time saved from repeatedly computing SSSPs.

8.2 Future Research

It is still a relatively new research area to solve the shortest paths in nearly acyclic graphs. There are possibilities of further improvement on some of the new algorithms presented in this thesis, and on some existing algorithms published by other researchers. Readers can find more information in Saunders' PhD thesis submitted in 2004 [21].

First of all, there is potential to develop an intelligent algorithm that can recognize the complexity of sc-components in a sub-graph, and then decide if a higher order sc-component decomposition is needed for the sub-graph. This can improve the performance of algorithms developed under *higher-order decomposition* framework. The *higher-order* approach presented in the first part of this research is heavily affected by the number of times that a graph is decomposed. That is, the decomposition time of this approach may by too costly for the overall efficiency of this approach for solving the SSSP problem. As a result, the new algorithm based on this approach can

only outperform other algorithms for some artificial graphs. But for totally random graphs, it perform worse than most of the existing SSSP algorithms. When the parameter ρ is decided from the second order or a higher order decomposition, it is dynamically defined similar to the parameter t in [1].

Another potential improvement is on *multi*-dominator sets. It is open whether we can compute a 2-dominator set or a 1-2-dominator set in $O(m)$ time. The number of triggers in the 2-dominator set or the 1-2-dominator set is about twenty percent less than that in the 1-dominator set. Using *multi*-dominator sets for solving the SSSP problem can gain some efficiency on sparse graphs. However, as discussed in Chapter VI, *multi*-dominator algorithms using the framework of 1-dominator decomposition are not efficient. The newly introduced 1-2-dominator framework has improved that in some aspects. But the decomposition time of the new algorithm for the 1-2-dominator set is still not linear. If we can determine the *multi*-dominator set in linear time, $O(m)$ time or close to $O(m)$ time, then the *multi*-dominator set immediately becomes practical and can outperform the 1-dominator set.

Overall, there is still some potential to further contribute to this research area. The theoretical research in this area enriches the knowledge of efficient shortest path computation. This may open a door to developing new algorithms that improve the classic time complexity $O(m + n\log n)$ for all kinds of graph. If someday people come across the shortest path problems on nearly acyclic graphs in the real world, the specialized shortest path algorithms developed in this thesis may contribute the practical benefits to the society.

References

[1] Abuaiadh D. and Kingston J. H., An Efficient Algorithm for the Shortest Path Problem. Tech. rep., Basser Department of Computer Science, University of Sydney, Australia, 1994, Technical Report 93-473.

[2] Abuaiadh D. and Kingston J. H., Are Fibonacci heaps optimal? ISAAC (1994) 442-450.

[3] Abuaiadh D. and Kingston J. H., Efficient Shortest Path Algorithms By Graph Decomposition. Tech. rep., Basser Department of Computer Science, University of Sydney, Australia, 1994, Technical Report 94-475.

[4] Alfred V. Aho, John E. Hopcroft, Jeffery D. Ullman, The Design and Analysis of Computer Algorithms. *Addison-Wesley*, 1974.

[5] Algorithm. http://en.wikipedia.org/wiki/Algorithm.

[6] Alan Gibbons. Algorithm Graph Theory. *Cambridge University Press* (1985).

[7] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. Introduction to Algorithms, 2nd ed. *MIT Press*, 2001.

[8] Dantzig G. B., On the shortest route through a network. *Management Science* 6 (1960), 187-190.

[9] Dijkstra, E. W., A note on two problems in connexion with graphs. *Numerische Mathematik* 1 (1959), 269-271.

[10] Fedman, M. L. and R. E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM* 34 (1987), 569-615.

[11] Floyd, R. W. Algorithm 97: Shortest path. *Communications of the ACM* 5 (1962), 345.

[12] Greg, N. Frederickson, Fast algorithm for shortest path in planar graph, with applications. *SIAM Journal on Computing* 16 (1987), 1004-1022.

[13] Fibonacci Heap. http://en.wikipedia.org/wiki/Fibonacci-heap

[14] Golumbic, Martin Charles. Algorithmic Graph Theory And Perfect Graphs. *Academic Press* (1980).

[15] Goldberg, A. V. Shortest path algorithms: Engineering aspects. *Lecture Notes in Computer Science* 2223 (2001), 502-513.

[16] Hagerup, T., Improved shortest paths on the word RAM. *In Proc. 27th Int'l Colloq. on Automata, Language, and Programming (ICALP)* (2000). Vol.1853 of Lecture Notes in Computer Science, 61-72.

[17] Johnson, D. B. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM* 24, 1 (1977), 1-13.

[18] Noshita, K. A theorem on the expected complexity of Dijkstra's shortest path algorithm. *Journal of Algorithms* 6, 3 (1985), 400-408.

[19] *P* And *NP* Problems. http://en.wikipedia.org/wiki/Complexity_classes_P_and_NP.

[20] Saunders, S. Takaoka, T., Improved Shortest Path Algorithms for Nearly Acyclic Graphs. *Theoretical Computer Science* 293(3) (2003), 535-556.

[21] Saunders Shane, Improved Shortest Path Algorithms for Nearly Acyclic Graphs. PhD. thesis at the University of Canterbury, New Zealand (2004).

[22] Saunders, S. Takaoka, T., Efficient Algorithm for Solving Shortest Paths on Nearly Acyclic Directed Graphs. *CATS* 2005.

[23] Takaoka, T., Shortest Path Algorithm for Nearly Acyclic Directed Graphs. *Theoretical Computer Science* 203(1) (1998), 145-150.

[24] Takaoka, T., Theory of 2-3 Heaps. *Discrete Applied Mathematics* 126 (2003), 115-128.

[25] Tarjan R. E., Depth-first search and linear graph algorithms. *SIAM Journal on Computing* 1 (2) (1972), 146-160.

[26] Tarjan, R.E., Data Structures and Network Algorithms. *Society for Industrial and Applied Mathematics* (1983).

[27] Tarjan, R. E., Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods* 6, 2 (1985), 306-318.

[28] Tadao Takaoka, The Lecture Note on Basic Graph Algorithms (2004).

[29] Tian L. and Takaoka T., Improved Shortest Path Algorithms for Nearly Acyclic Directed Graphs. *CRPIT*, 2007, Vol. 62.

[30] Van Emde Boas, P., Kaas, R., and Zijlstra, E., Design and implementation of an efficient priority queue. *Math. Systems Theory* 10 (1977), 99-127.

Appendix A

Publication

The research contained in this thesis appeared at the Thirtieth Australasian Computer Science Conference (ACSC2007), Ballarat, Australia. A paper of the research is to be published through the Conferences in Research and Practice in Information Technology (CRPIT), Vol. 62. The reference to the article is duplicated here:

[**28**] Tian L. and Takaoka T., Improved Shortest Path Algorithms for Nearly Acyclic Directed Graphs. *CRPIT*, 2007, Vol. 62.